Fundamentals of Software Engineering

Fourth Edition

RAJIB MALL
Professor
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

PHI Learning Private Limited
Delhi-110 092
2014

FUNDAMENTALS OF SOFTWARE ENGINEERING, Fourth Edition
Rajib Mall

ISBN-978-81-203-4898-1

Thirty-first Printing (Fourth Edition) . . . . . April, 2014

Published by Asoke K. Ghosh, PHI Learning Private Limited, Rimjhim House, 111, Patparganj Industrial Estate, Delhi-110092 and Printed by Rajkamal Electric Press, Plot No. 2, Phase IV, HSIDC, Kundli-131028, Sonepat, Haryana.

To
Bapa, Maa,
and
my beloved wife Prabina

CONTENTS

List of Figures xv
Preface xix
Preface to the First Edition xxi
1. INTRODUCTION 1–32
1.1 Evolution—From an Art Form to an Engineering Discipline 3
1.1.1 Evolution of an Art into an Engineering Discipline 3
1.1.2 Evolution Pattern for Engineering Disciplines 4
1.1.3 A Solution to the Software Crisis 5
1.2 Software Development Projects 6
1.2.1 Types of Software Development Projects 7
1.2.2 Software Projects Being Undertaken by Indian Companies 8
1.3 Exploratory Style of Software Development 9
1.3.1 Perceived Problem Complexity: An Interpretation Based on Human Cognition Mechanism 11
1.3.2 Principles Deployed by Software Engineering to Overcome Human Cognitive Limitations 14
1.4 Emergence of Software Engineering 17
1.4.1 Early Computer Programming 17
1.4.2 High-level Language Programming 18
1.4.3 Control Flow-based Design 18
1.4.4 Data Structure-oriented Design 22
1.4.5 Data Flow-oriented Design 22
1.4.6 Object-oriented Design 23
1.4.7 What Next? 24
1.4.8 Other Developments 25
1.5 Notable Changes in Software Development Practices 25
1.6 Computer Systems Engineering 27
Summary 28
Exercises 29
2. Software Life Cycle Models 33–84
2.1 A Few Basic Concepts 34

2.2 Waterfall Model and its Extensions 38
2.2.1 Classical Waterfall Model 38
2.2.2 Iterative Waterfall Model 46
2.2.3 V-Model 50
2.2.4 Prototyping Model 52
2.2.5 Incremental Development Model 55
2.2.6 Evolutionary Model 57
2.3 Rapid Application Development (RAD) 59
2.3.1 Working of RAD 60
2.3.2 Applicability of RAD Model 60
2.3.3 Comparison of RAD with Other Models 62
2.4 Agile Development Models 62
2.4.1 Essential Idea behind Agile Models 64
2.4.2 Agile versus Other Models 65
2.4.3 Extreme Programming Model 66
2.4.4 Scrum Model 69
2.5 Spiral Model 69
2.5.1 Phases of the Spiral Model 71
2.6 A Comparison of Different Life Cycle Models 72
2.6.1 Selecting an Appropriate Life cycle Model for a Project 73

Summary 74

Exercises 75

3. SOFTWARE PROJECT MANAGEMENT 85–153

3.1 Software Project Management Complexities 86
3.2 Responsibilities of a Software Project Manager 87
3.2.1 Job Responsibilities for Managing Software Projects 87
3.2.2 Skills Necessary for Managing Software Projects 88
3.3 Project Planning 89
3.3.1 Sliding Window Planning 90
3.3.2 The SPMP Document of Project Planning 90
3.4 Metrics for Project Size Estimation 92
3.4.1 Lines of Code (LOC) 92
3.4.2 Function Point (FP) Metric 94
3.5 Project Estimation Techniques 99
3.5.1 Empirical Estimation Techniques 99
3.5.2 Heuristic Techniques 99
3.5.3 Analytical Estimation Techniques 100

3.6 Empirical Estimation Techniques 100
3.6.1 Expert Judgement 100
3.6.2 Delphi Cost Estimation 101
3.7 COCOMO—A Heuristic Estimation Technique 101
3.7.1 Basic COCOMO Model 102
3.7.2 Intermediate COCOMO 107
3.7.3 Complete COCOMO 108
3.7.4 COCOMO 2 109
3.8 Halstead's Software Science—An Analytical Technique 112
3.8.1 Length and Vocabulary 113
3.8.2 Program Volume 113
3.8.3 Potential Minimum Volume 113
3.8.4 Effort and Time 114
3.8.5 Length Estimation 114
3.9 Staffing Level Estimation 116
3.9.1 Norden's Work 116
3.9.2 Putnam's Work 117
3.9.3 Jensen's Model 119
3.10 Scheduling 119
3.10.1 Work Breakdown Structure 121
3.10.2 Activity Networks 122
3.10.3 Critical Path Method (CPM) 124
3.10.4 PERT Charts 126
3.10.5 Gantt Charts 128
3.11 Organisation and Team Structures 129
3.11.1 Organisation Structure 129
3.11.2 Team Structure 132
3.12 Staffing 135
3.13 Risk Management 136
3.13.1 Risk Identification 137
3.13.2 Risk Assessment 138
3.13.3 Risk Mitigation 138
3.14 Software Configuration Management 140
3.14.1 Necessity of Software Configuration Management 140
3.14.2 Configuration Management Activities 142
3.15 Miscellaneous Plans 144

Summary 144

Exercises 145

4. REQUIREMENTS ANALYSIS AND SPECIFICATION 154–200

4.1 Requirements Gathering and Analysis 155
4.1.1 Requirements Gathering 156
4.1.2 Requirements Analysis 159
4.2 Software Requirements Specification (SRS) 161
4.2.1 Users of SRS Document 161
4.2.2 Why Spend Time and Resource to Develop an SRS Document? 162
4.2.3 Characteristics of a Good SRS Document 163
4.2.4 Attributes of Bad SRS Documents 164
4.2.5 Important Categories of Customer Requirements 165
4.2.6 Functional Requirements 167
4.2.7 How to Identify the Functional Requirements? 170
4.2.8 How to Document the Functional Requirements? 170
4.2.9 Traceability 173
4.2.10 Organisation of the SRS Document 173
4.2.11 Techniques for Representing Complex Logic 180
4.3 Formal System Specification 182
4.3.1 What is a Formal Technique? 183
4.3.2 Operational Semantics 184
4.4 Axiomatic Specification 186
4.5 Algebraic Specification 188
4.5.1 Auxiliary Functions 191
4.5.2 Structured Specification 192
4.6 Executable Specification and 4GL 193

Summary 193

Exercises 193

5. SOFTWARE DESIGN 201–222

5.1 Overview of the Design Process 201
5.1.1 Outcome of the Design Process 201
5.1.2 Classification of Design Activities 202
5.1.3 Classification of Design Methodologies 203
5.2 How to Characterise a Good Software Design? 204
5.2.1 Understandability of a Design: A Major Concern 205
5.3 Cohesion and Coupling 208

5.3.1 Classification of Cohesiveness 209
5.3.2 Classification of Coupling 211
5.4 Layered Arrangement of Modules 212
5.5 Approaches to Software Design 214
5.5.1 Function-oriented Design 214
5.5.2 Object-oriented Design 215

Summary 219

Exercises 219

6. FUNCTION-ORIENTED SOFTWARE DESIGN 223–275
6.1 Overview of SA/SD Methodology 224
6.2 Structured Analysis 225
6.2.1 Data Flow Diagrams (DFDs) 225
6.3 Developing the DFD Model of a System 229
6.3.1 Context Diagram 229
6.3.2 Level 1 DFD 231
6.3.3 Extending DFD Technique to Make it Applicable to Real-Time Systems 246
6.4 Structured Design 247
6.4.1 Transformation of a DFD Model into Structure Chart 248
6.5 Detailed Design 253
6.6 Design Review 253

Summary 254

Exercises 254

7. Object Modelling Using UML 276–334
7.1 Basic Object-Orientation Concepts 277
7.1.1 Basic Concepts 277
7.1.2 Class Relationships 281
7.1.3 How to Identify Class Relationships? 288
7.1.4 Other Key Concepts 289
7.1.5 Related Technical Terms 294
7.1.6 Advantages and Disadvantages of OOD 295
7.2 Unified Modelling Language (UML) 296
7.2.1 Origin of UML 296
7.2.2 Evolution of UML 298
7.3 UML Diagrams 300

7.4 Use Case Model 302
7.4.1 Representation of Use Cases 303
7.4.2 Why Develop the Use Case Diagram? 307
7.4.3 How to Identify the Use Cases of a System? 307
7.4.4 Essential Use Case versus Real Use Case 307
7.4.5 Factoring of Commonality among Use Cases 308
7.4.6 Use Case Packaging 310
7.5 Class Diagrams 311
7.6 Interaction Diagrams 318
7.7 Activity Diagram 320
7.8 State Chart Diagram 322
7.9 Postscript 323
7.9.1 Package, Component, and Deployment Diagrams 323
7.9.2 UML 2.0 325

Summary 327

Exercises 328

8. Object-Oriented Software Development **335–372**

8.1 Patterns 337
8.1.1 Basic Pattern Concepts 337
8.1.2 Types of Patterns 338
8.1.3 More Pattern Concepts 340
8.2 Some Common Design Patterns 341
8.3 An Object-Oriented Analysis and Design (OOAD) Methodology 349
8.3.1 Unified Process 349
8.3.2 Overview of The OOAD Methodology 350
8.3.3 Use Case Model Development 351
8.3.4 Domain Modelling 353
8.3.5 Identification of Entity Objects 357
8.3.6 Booch's Object Identification Method 357
8.3.7 Interaction Modelling 360
8.3.8 Class-Responsibility-Collaborator (CRC) Cards 360
8.4 Applications of the Analysis and Design Process 361
8.5 OOD Goodness Criteria 364

Summary 369

Exercises 369

9. USER INTERFACE DESIGN 373–396

9.1 Characteristics of a Good User Interface 374
9.2 Basic Concepts 376
9.2.1 User Guidance and On-line Help 376
9.2.2 Mode-based versus Modeless Interface 377
9.2.3 Graphical User Interface (GUI) versus Text-based User Interface 377
9.3 Types of User Interfaces 378
9.3.1 Command Language-based Interface 378
9.3.2 Menu-based Interface 379
9.3.3 Direct Manipulation Interfaces 381
9.4 Fundamentals of Component-based GUI Development 381
9.4.1 Window System 382
9.4.2 Types of Widgets 385
9.4.3 An Overview of X-Window/MOTIF 386
9.4.4 X Architecture 387
9.4.5 Size Measurement of a Component-based GUI 388
9.5 A User Interface Design Methodology 388
9.5.1 Implications of Human Cognition Capabilities on User Interface Design 389
9.5.2 A GUI Design Methodology 389

Summary 393

Exercises 394

10. Coding and Testing 397–456

10.1 Coding 398
10.1.1 Coding Standards and Guidelines 399
10.2 Code Review 400
10.2.1 Code Walkthrough 401
10.2.2 Code Inspection 402
10.2.3 Clean Room Testing 403
10.3 Software Documentation 403
10.3.1 Internal Documentation 404
10.3.2 External Documentation 404
10.4 Testing 405
10.4.1 Basic Concepts and Terminologies 406
10.4.2 Testing Activities 410

10.4.3 Why Design Test Cases? 411
10.4.4 Testing in the Large versus Testing in the Small 412
10.5 Unit Testing 413
10.6 Black-box Testing 413
10.6.1 Equivalence Class Partitioning 414
10.6.2 Boundary Value Analysis 415
10.6.3 Summary of the Black-box Test Suite Design Approach 417
10.7 White-Box Testing 417
10.7.1 Basic Concepts 417
10.7.2 Statement Coverage 419
10.7.3 Branch Coverage 419
10.7.4 Multiple Condition Coverage 420
10.7.5 Path Coverage 421
10.7.6 McCabe's Cyclomatic Complexity Metric 423
10.7.7 Data Flow-based Testing 425
10.7.8 Mutation Testing 426
10.8 Debugging 427
10.8.1 Debugging Approaches 427
10.8.2 Debugging Guidelines 428
10.9 Program Analysis Tools 428
10.9.1 Static Analysis Tools 428
10.9.2 Dynamic Analysis Tools 429
10.10 Integration Testing 430
10.10.1 Phased versus Incremental Integration Testing 431
10.11 Testing Object-Oriented Programs 432
10.11.1 What is a Suitable Unit for Testing Object-Oriented Programs? 432
10.11.2 Do Various Object-Orientation Features Make Testing Easy? 433
10.11.3 Why are Traditional Techniques Considered Not Satisfactory for Testing Object-Oriented Programs? 434
10.11.4 Grey-Box Testing of Object-Oriented Programs 434
10.11.5 Integration Testing of Object-oriented Programs 435
10.12 System Testing 435
10.12.1 Smoke Testing 436
10.12.2 Performance Testing 436
10.12.3 Error Seeding 438
10.13 Some General Issues Associated with Testing 439

Summary 440

Exercises 440

11. Software Reliability and Quality Management **457–484**

11.1 Software Reliability 458
11.1.1 Hardware versus Software Reliability 459
11.1.2 Reliability Metrics of Software Products 460
11.1.3 Reliability Growth Modelling 462
11.2 Statistical Testing 463
11.2.1 Steps in Statistical Testing 463
11.3 Software Quality 464
11.4 Software Quality Management System 465
11.4.1 Evolution of Quality Systems 466
11.4.2 Product Metrics versus Process Metrics 467
11.5 ISO 9000 467
11.5.1 What is ISO 9000 Certification? 467
11.5.2 ISO 9000 for Software Industry 468
11.5.3 Why Get ISO 9000 Certification? 469
11.5.4 How to Get ISO 9000 Certification? 469
11.5.5 Summary of ISO 9001 Requirements 470
11.5.6 Salient Features of ISO 9001 Requirements 472
11.5.7 ISO 9000-2000 472
11.5.8 Shortcomings of ISO 9000 Certification 472
11.6 SEI Capability Maturity Model 473
11.6.1 Comparison between ISO 9000 Certification and SEI/CMM 476
11.6.2 Is SEI CMM Applicable to Small Organisations? 476
11.6.3 Capability Maturity Model Integration (CMMI) 477
11.7 Few Other Important Quality Standards 477
11.7.1 Software Process Improvement and Capability Determination (SPICE) 477
11.7.2 Personal Software Process (PSP) 477
11.8 Six Sigma 479

Summary 480

Exercises 481

12. Computer Aided Software Engineering **485–493**

12.1 Case and its Scope 485
12.2 Case Environment 485
12.2.1 Benefits of CASE 487
12.3 CASE Support in Software Life Cycle 487
12.3.1 Prototyping Support 487
12.3.2 Structured Analysis and Design 488
12.3.3 Code Generation 488
12.3.4 Test Case Generator 489
12.4 Other Characteristics of Case Tools 489
12.4.1 Hardware and Environmental Requirements 489
12.4.2 Documentation Support 489
12.4.3 Project Management 490
12.4.4 External Interface 490
12.4.5 Reverse Engineering Support 490
12.4.6 Data Dictionary Interface 490
12.4.7 Tutorial and Help 490
12.5 Towards Second Generation CASE Tool 490
12.6 Architecture of a Case Environment 491
Summary 492
Exercises 492

13. Software Maintenance 494–502

13.1 Characteristics of Software Maintenance 494
13.1.1 Characteristics of Software Evolution 495
13.1.2 Special Problems Associated with Software Maintenance 496
13.2 Software Reverse Engineering 496
13.3 Software Maintenance Process Models 497
13.4 Estimation of Maintenance Cost 500
Summary 501
Exercises 501

14. SOFTWARE REUSE 503–512

14.1 What can be Reused? 503
14.2 Why Almost No Reuse So Far? 504
14.3 Basic Issues in any Reuse Program 504
14.4 A Reuse Approach 505
14.4.1 Domain Analysis 505
14.4.2 Component Classification 506
14.4.3 Searching 507

14.4.4 Repository Maintenance 507
14.4.5 Reuse without Modifications 508
14.5 Reuse at Organisation Level 508
14.5.1 Current State of Reuse 510
Summary 510
Exercises 511

15. EMERGING TRENDS 513–525

15.1 Client-Server Software 514
15.2 Client-server Architectures 516
15.3 CORBA 518
15.3.1 CORBA Reference Model 518
15.3.2 CORBA ORB Architecture 519
15.3.3 CORBA Implementations 521
15.3.4 Software Development in CORBA 521
15.4 COM/DCOM 522
15.4.1 COM 522
15.4.2 DCOM 522
15.5 Service-Oriented Architecture (SOA) 522
15.5.1 Service-oriented Architecture (SOA): Mitty Gritty 523
15.6 Software as a Service (SaaS) 524
Summary 524
Exercises 525

ReferenWces 527–530

Index 531–534

LIST OF FIGURES

1.1 Evolution of technology with time 4
1.2 Relative changes of hardware and software costs over time 5
1.3 Exploratory program development 9
1.4 Increase in development time and effort with problem size 10
1.5 Human cognition mechanism model 12
1.6 Schematic representation 14
1.7 An abstraction hierarchy classifying living organisms 16
1.8 An example of (a) Unstructured program (b) Corresponding structured program 19
1.9 Control flow graphs of the programs in Figures 1.8(a) and (b) 19
1.10 CFG of a program having too many GO TO statements 20
1.11 Data flow model of a car assembly plant 23
1.12 Evolution of software design techniques 24
1.13 Computer systems engineering 28

2.1 Classical waterfall model 39
2.2 Relative effort distribution among different phases of a typical product 40
2.3 Iterative waterfall model 46
2.4 Distribution of effort for various phases in the iterative waterfall model 48
2.5 V-model 51
2.6 Prototyping model of software development 54
2.7 Incremental software development 55
2.8 Incremental model of software development 56
2.9 Evolutionary model of software development 58
2.10 Spiral model of software development 70

3.1 Precedence ordering among planning activities 90
3.2 System function as a mapping of input data to output data 95
3.3 Person-month curve 104
3.4 Effort versus product size 105
3.5 Development time versus size 106
3.6 Rayleigh curve 116

3.7 Work breakdown structure of an MIS problem 121
3.8 Activity network representation of the MIS problem 123
3.9 AoN for MIS problem with (ES,EF) 125
3.10 AoN of MIS problem with (LS,LF) 126
3.11 PERT chart representation of the MIS problem 127
3.12 Gantt chart representation of the MIS problem 128
3.13 Schematic representation of the functional and project organisation 130
3.14 Matrix organisation 132
3.15 Chief programmer team structure 133
3.16 Democratic team structure 134
3.17 Mixed team structure 135
3.18 Reserve and restore operation in configuration control 143
4.1 The black-box view of a system as performing a set of functions 164
4.2 User and system interactions in high-level functional requirement. 169
4.3 Decision Tree for LMS 181
5.1 The design process 201
5.2 Two design solutions to the same problem 207
5.3 Classification of cohesion 209
5.4 Examples of cohesion 210
5.5 Classification of coupling 212
5.6 Examples of good and poor control abstraction 214
6.1 Structured analysis and structured design methodology 224
6.2 Symbols used for designing DFDs 226
6.3 Synchronous and asynchronous data flow 227
6.4 DFD model of a system consists of a hierarchy of DFDs and a single data dictionary 230
6.5 An example showing balanced decomposition 233
6.6 It is incorrect to show control information on a DFD 234
6.7 Illustration of how to avoid data cluttering 235
6.8 Context diagram, level 1, and level 2 DFDs for Example 6.1 236
6.9 Context diagram and level 1 DFDs for Example 6.2 238
6.10 Context diagram for Example 6.3 239
6.11 Level 1 diagram for Example 6.3 240
6.12 Level 2 diagram for Example 6.3 240
6.13 Context diagram for Example 6.4 242
6.14 Level 1 DFD for Example 6.4 242

6.15 Context diagram for Example 6.5 244
6.16 Level 1 DFD for Example 6.5 245
6.17 Level 2 DFD for Example 6.5 245
6.18 Examples of properly and poorly layered designs 248
6.19 Structure chart for Example 6.6 250
6.20 Structure chart for Example 6.7 251
6.21 Structure chart for Example 6.8 252
6.22 Structure chart for Example 6.9 252
6.23 Structure chart for Example 6.10 253
7.1 Important concepts used in the object-oriented approach 277
7.2 A model of an object 279
7.3 Library information system example 282
7.4 An example of multiple inheritance 284
7.5 Example of (a) binary (b) ternary (c) unary association 285
7.6 Example of aggregation relationship 287
7.7 An example of an abstract class 288
7.8 Schematic representation of the concept of encapsulation 290
7.9 Circle class with overloaded create method 292
7.10 Class hierarchy of geometric objects 293
7.11 Traditional code versus object-oriented code incorporating the dynamic binding feature 293
7.12 Schematic representation of the impact of different object modelling techniques on UML 297
7.13 Evolution of UML 298
7.14 Different types of diagrams and views supported in UML 301
7.15 Use case model for Example 7.2 305
7.16 Use case model for Example 7.3 306
7.17 Representation of use case generalisation 308
7.18 Representation of use case inclusion 309
7.19 Example of use case inclusion 309
7.20 Example of use case extension 310
7.21 Hierarchical organisation of use cases 311
7.22 Use case packaging 312
7.23 Different representations of the LibraryMember class 313
7.24 Association between two classes 314
7.25 Representation of aggregation 315
7.26 Representation of composition 315

7.27 Representation of the inheritance relationship 316
7.28 Representation of dependence between classes 317
7.29 Different representations of a LibraryMember object 317
7.30 Sequence diagram for the renew book use case 319
7.31 Collaboration diagram for the renew book use case 320
7.32 Activity diagram for student admission procedure at IIT 321
7.33 State chart diagram for an order object 323
7.34 An example package diagram 324
7.35 Anatomy of a combined fragment in UML 2.0 326
7.36 An example sequence diagram showing a combined fragment in UML 2.0 327
8.1 Expert pattern: (a) Class diagram (b) Collaboration diagram 342
8.2 Service invocation with and without using a facade class 344
8.3 Interaction diagram for the observer pattern 345
8.4 Class structure for the MVC pattern 346
8.5 Interaction model for the MVC pattern 346
8.6 Interaction model of the publish-subscribe pattern 347
8.7 A schematic representation of the publish-subscribe pattern 348
8.8 Unified process model 350
8.9 An object-oriented analysis and design process 351
8.10 A typical realisation of a use case through the collaboration of boundary, controller, and entity objects 355
8.11 CRC card for the `BookRegister` class 361
8.12 Use case model for Example 8.2 362
8.13 (a) Initial domain model (b) Refined domain model for Example 8.2 362
8.14 Sequence diagram for the play move use case of Example 8.2 363
8.15 Class diagram for Example 8.2 364
8.16 Use case model for Example 8.3 364
8.17 (a) Initial domain model (b) Refined domain model for Example 8.3 365
8.18 Sequence diagram for the select winner list use case of Example 8.3 366
8.19 Sequence diagram for the register customer use case of Example 8.3 366
8.20 Sequence diagram for the register sales use case of Example 8.3 367
8.21 Refined sequence diagram for the register sales use case of Example

8.3 367
8.22 Class diagram for Example 8.3 368
9.1 Font size selection using scrolling menu 380
9.2 Example of walking menu 380
9.3 Window with client and user areas marked 382
9.4 Window management system 384
9.5 Network-independent GUI 386
9.6 Architecture of the X system 387
9.7 Decomposition of a task into subtasks 391
9.8 State chart diagram for an order object 391
10.1 A simplified view of program testing 406
10.2 Testing process 411
10.3 Unit testing with the help of driver and stub modules 413
10.4 Equivalence classes for Example 10.6 415
10.5 CFG for (a) sequence, (b) selection, and (c) iteration type of constructs 416
10.6 Illustration of stronger, weaker, and complementary testing strategies 418
10.7 Control flow diagram of an example program 422
10.8 Module C Sequentially Integrates Modules A and B 455
11.1 Change in failure rate of a product 460
11.2 Step function model of reliability growth 462
11.3 Evolution of quality system and corresponding shift in the quality paradigm 466
11.4 A schematic representation of PSP 478
11.5 Levels of PSP 479
12.1 A CASE environment 486
12.2 Architecture of a modern CASE environment 491
13.1 A process model for reverse engineering 497
13.2 Cosmetic changes carried out before reverse engineering 497
13.3 Maintenance process model 1 499
13.4 Maintenance process model 2 499
13.5 Empirical estimation of maintenance cost versus percentage rework 500
14.1 Improving reusability of a component by using a portability interface 510

15.1 Two-tier and three-tier client-server architectures 517
15.2 CORBA reference model 518
15.3 CORBA ORB architecture 520

PREFACE

The revision to this book had become necessary on account of the rapid advancements that have taken place in software engineering techniques and practices since the last edition was written. In this book, almost all the chapters have been enhanced. Also, many objective type questions have been included in almost every chapter. This book has taken shape over the two decades decades while teaching the **Software Engineering** subject to the undergraduate and postgraduate students at IIT, Kharagpur.

While teaching to the students, I had acutely felt the necessity of a book that treats all the important topics in software engineering, including the important recent advancements in a coherent framework and at the same time deals the topics from the perspective of the practising engineer. A large portion of the text is based on my own practical experience which I gained while working on software development projects in several organizations.

This book is designed to serve as a text book for one semester course on software engineering for undergraduate students by excluding the star marked sections in different chapters. The topics on Halsteads software science, Software reuse, and Formal specification can be omitted for a basic study of the subject, if so desired by the teacher. However, these topics should be included in a post-graduate level course. For postgraduate students, this text book may be supplemented with some additional topics.

The students intending to go through this book must be familiar with at least one high level programming and one low level programming language. They are also expected to possess basic ideas about operating systems, systems programming, compiler writing, and computer architecture issues. Experience in writing large-sized programs would be very helpful in grasping some of the important concepts discussed in this book. The emphasis of this book is to illustrate the important concepts through small examples rather than through a single large running example. I have intentionally selected

the former approach as I believe that this would make it easier to illustrate several subtle and important concepts through appropriate small examples. It would have been very difficult to illustrate all these concepts through a single running example.

The layout of the chapters has been guided by the sequence of activities undertaken during the life of a software product. However, since the project management activity is spread over all phases, I thought that it is necessary to discuss these as early in the book

book as possible. Software project management has been discussed in Chapter 3. However,

while teaching from this book, I prefer to teach the project management topic after the Chapter 11, since that way I am able to give the design assignments to the students early and they get sufficient time to complete them.

In the text, I have taken the liberty to use he/his to actually mean both the genders.

This has been done only to increase the readability of the writing rather than with intent of

any bias.

The power-point slides to teach the book as well as the solution manual can be obtained

either from the publisher or by sending me an e-mail.

Typographical and other errors and comments should be reported to me at:

rajib@cse.iitkgp.ernet.in

or at my following postal address.

RAJIB MALL

Professor

Department of Computer Science and Engineering

Indian Institute of Technology Kharagpur

PREFACE TO THE FIRST EDITION

This book is designed as a textbook on software engineering for undergraduate students in computer science. Software engineering is a fast developing field. While teaching the subject at the Indian Institute of Technology Kharagpur, I felt the need for organizing a textbook that gives a coherent account of all the state-of-the-art topics and at the same time presents these topics from the viewpoint of practising engineers. A portion of the text is, therefore, based on my own practical experience, gained while working on software development projects in several industries.

The book starts with a comprehensive introduction to software engineering, including some important life cycle models. Chapter 2 presents and discusses techniques and concepts of software project management. This chapter encompasses all phases of software development that are considered crucial to the success of software projects. Chapter 3 focuses on requirements analysis and specification. In this chapter, different approaches to formal requirements specification and essential features of algebraic specifications as a formal specification technique are explored. Chapter 4 highlights some important facets of software design. In Chapter 5, the methodology of Structured Analysis/Structured Design (SA/SD) in relation to traditional function-oriented design. Chapter 7 brings out some basic aspects, techniques and methods pertaining to user interface design. Significant progress has been made in this field and it is important for students to know the various issues involved in a good user interface design. Chapter 8 discusses coding and unit testing techniques. Integration and system testing techniques are elaborately described in Chapter 9. These are the main quality control activities. Chapter 10 is, therefore, exclusively devoted to software quality assurance aspects, ISO 9000 and software reliability models, as these are considered necessary to expose students to basic quality concepts as part of a software engineering course. Finally, in Chapter 11, the student has been introduced to general concepts to CASE tools, without going into specifics of any particular CASE tool.

The students using this textbook should be proficient at least in one high level and low level programming language each. They should also possess basic knowledge of operating systems, systems programming, compiler writing, and computer architecture.

The emphasis in this book is to illustrate the important concepts through several small examples rather than a single large running example. The book also contains many exercises at the end of each chapter aimed at reinforcing the knowledge of principles

and techniques of software engineering.

I do hope fervently that the students will find this text both stimulating and useful.

ACKNOWLEDGEMENTS

Many persons have contributed to make this book a reality. I would especially like to express my appreciation to Prof. L.M. Patnaik for his unstinted support and encouragement. I would also like to thank Prof. Sunil Sarangi, Dean (CEP) for his guidance throughout the preparation of the manuscript. Thanks are also due to Prof. Ajit Pal, the present Head of the Department, and all my colleagues at the Computer Science and Engineering Department of IIT Kharagpur for their helpful comments and suggestions. I express my special thanks to Prof. P.K.J. Mahapatra of IEM Department for his help during the final preparation of the manuscript.

I acknowledge the help and cooperation received from all the staff members of the Computer Science and Engineering Department of IIT Kharagpur.

I would like to acknowledge the financial assistance provided by IIT Kharagpur for the preparation of the manuscript and I wish to thank the numerous B.Tech and M.Tech students whose enthusiastic participation in classroom discussions helped me to present many ideas and concepts, as discussed in this book, with greater clarity.

Finally, I wish to express my sincere thanks to all my family members for their moral support. In particular, I thank my parents, Sanjib, Kanika Bhabhi, Sudip, Shivani, Sonali and Amitabha, my parents-in-law and GUGLOO. I am grateful to my wife Prabina for her constant encouragement.

RAJIB MALL

Preface to the First Edition

Chapter 1

INTRODUCTION

Commercial usage of computers now spans the last sixty years. Computers were very slow in the initial years and lacked sophistication. Since then, their computational power and sophistication increased rapidly, while their prices dropped dramatically. To get an idea of the kind of improvements that have occurred to computers, consider the following analogy. If similar improvements could have occurred to aircrafts, now personal mini-airplanes should have become available, costing as much as a bicycle, and flying at over 1000 times the speed of the supersonic jets. To say it in other words, the rapid strides in computing technologies are unparalleled in any other field of human endeavour.

Let us now reflect the impact of the astounding progress made to the hardware technologies on the software. The more powerful a computer is, the more sophisticated programs can it run. Therefore, with every increase in the raw computing capabilities of computers, software engineers have been called upon to solve increasingly larger and complex problems, and that too in cost-effective and efficient ways. Software engineers have coped up with this challenge by innovating and building upon their past programming experiences.

The innovations and past experiences towards writing good quality programs cost-effectively, have contributed to the emergence of the software engineering discipline.

Let us now examine the scope of the software engineering discipline more closely.

What is software engineering?

A popular definition of software engineering is: "A systematic collection of good program development practices and techniques". Good program

development techniques have resulted from research innovations as well as from the lessons learnt by programmers through years of programming experiences. An alternative definition of software engineering is: "An *engineering approach* to develop software". Based on these two point of views, we can define software engineering as follows:

> Software engineering discusses systematic and cost-effective techniques for software development. These techniques help develop software using an engineering approach.

Let us now try to figure out what exactly is meant by an *engineering approach* to develop software. We explain this using an analogy. Suppose you have asked a petty contractor to build a small house for you. Petty contractors are not really experts in house building.

They normally carry out minor repair works and at most undertake very small building works such as the construction of boundary walls. Now faced with the task of building a complete house, your petty contractor would draw upon all his knowledge regarding house building. Yet, he may often be clueless regarding what to do. For example, he might not know the optimal proportion in which cement and sand should be mixed to realise sufficient strength for supporting the roof. In such situations, he would have to fall back upon his intuitions. He would normally succeed in his work, if the house you asked him to construct is sufficiently small. Of course, the house constructed by him may not look as good as one constructed by a professional, may lack proper planning, and display several defects and imperfections. It may even cost more and take longer to build.

Now, suppose you entrust your petty contractor to build a large 50-storeyed commercial complex for you. He might exercise prudence, and politely refuse to undertake your request. On the other hand, he might be ambitious and agree to undertake the task. In the later case, he is sure to fail. The failure might come in several forms—the building might collapse during the construction stage itself due to his ignorance of the basic theories concerning the strengths of materials; the construction might get unduly delayed, since he may not prepare proper estimates and detailed plans regarding the types and quantities of raw materials required, the times at which these are required, etc. In short, to be successful in constructing a building of large magnitude, one needs a good understanding of various civil and architectural engineering techniques such as analysis, estimation, prototyping, planning, designing, and testing. Similar is the case with the software development

projects. For sufficiently small-sized problems, one might proceed according to one's intuition and succeed; though the solution may have several imperfections, cost more, take longer to complete, etc. But, failure is almost certain, if one without a sound understanding of the software engineering principles undertakes a large-scale software development work.

Is software engineering a science or an art?

Several people hold the opinion that writing good quality programs is an art. In this context, let us examine whether software engineering is really a form of art or is it akin to other engineering disciplines. There exist several fundamental issues that set engineering disciplines such as software engineering and civil engineering apart from both science and arts disciplines. Let us now examine where software engineering stands based on an investigation into these issues:

- Just as any other engineering discipline, software engineering makes heavy use of the knowledge that has accrued from the experiences of a larges number o f practitioners. These past experiences have been systematically organised and wherever possible theoretical basis to the empirical observations have been provided. Whenever no reasonable theoretical justification could be provided, the past experiences have been adopted as rule of thumb. In contrast, all scientific solutions are constructed through rigorous application of provable principles.
- As is usual in all engineering disciplines, in software engineering several conflicting goals are encountered while solving a problem. In such situations, several alternate solutions are first proposed. An appropriate solution is chosen out of the candidate solutions based on various trade-offs that need to be made on account of issues of cost, maintainability, and usability. Therefore, while arriving at the final solution, several iterations and are possible.
- Engineering disciplines such as software engineering make use of only well-understood and well-documented principles. Art, on the other hand, is often based on making subjective judgement based on qualitative attributes and using ill-understood principles.

From the above, we can easily infer that software engineering is in many ways similar to other engineering disciplines such as civil engineering or electronics engineering.

1.1 EVOLUTION—FROM AN ART FORM TO A N ENGINEERING DISCIPLINE

In this section, we review how starting from an esoteric art form, the software engineering discipline has evolved over the years.

1.1.1 Evolution of an Art into an Engineering Discipline

Software engineering principles have evolved over the last sixty years with contributions from numerous researchers and software professionals. Over the years, it has emerged from a pure art to a craft, and finally to an engineering discipline.

The early programmers used an *ad hoc* programming style. This style of program development is now variously being referred to as *exploratory*, *build and fix*, and *code and fix* styles.

In a build and fix style, a program is quickly developed without making any specification, plan, or design. The different imperfections that are subsequently noticed are fixed.

The exploratory programming style is an informal style in the sense that there are no set rules or recommendations that a programmer has to adhere to—every programmer himself evolves his own software development techniques solely guided by his own intuition, experience, whims, and fancies. The exploratory style comes naturally to all first time programmers. Later in this chapter we point out that except for trivial problems, the exploratory style usually yields poor quality and unmaintainable code and also makes program development very expensive as well as time-consuming.

As we have already pointed out, the build and fix style was widely adopted by the programmers in the early years of computing history. We can consider the exploratory program development style as an art—since this style, as is the case with any art, is mostly guided by intuition. There are many stories about programmers in the past who were like proficient artists and could write good programs using an essentially build and fix model and some esoteric knowledge. The bad programmers were left to wonder how could some programmers effortlessly write elegant and correct programs each time. In contrast, the programmers working in modern software industry rarely make use of any esoteric knowledge and develop software by applying some well-understood principles.

1.1.2 Evolution Pattern for Engineering Disciplines

If we analyse the evolution of the software development styles over the last sixty years, we can easily notice that it has evolved from an esoteric art form to a craft form, and then has slowly emerged as an engineering discipline. As a matter of fact, this pattern of evolution is not very different from that seen in other engineering disciplines. Irrespective of whether it is iron making, paper making, software development, or building construction; evolution of technology has followed strikingly similar patterns. This pattern of technology development has schematically been shown in Figure 1.1. It can be seen from Figure 1.1 that every technology in the initial years starts as a form of art. Over time, it graduates to a craft and finally emerges as an engineering discipline. Let us illustrate this fact using an example. Consider the evolution of the iron making technology. In ancient times, only a few people knew how to make iron. Those who knew iron making, kept it a closely-guarded secret. This esoteric knowledge got transferred from generation to generation as a family secret. Slowly, over time technology graduated from an art to a craft form where tradesmen shared their knowledge with their apprentices and the knowledge pool continued to grow. Much later, through a systematic organisation and documentation of knowledge, and incorporation of scientific basis, modern steel making technology emerged. The story of the evolution of the software engineering discipline is not much different. As we have already pointed out, in the early days of programming, there were good programmers and bad programmers. The good programmers knew certain principles (or tricks) that helped them write good programs, which they seldom shared with the bad programmers. Program writing in later years was akin to a craft. Over the next several years, all good principles (or tricks) that were organised into a body of knowledge that forms the discipline of software engineering.

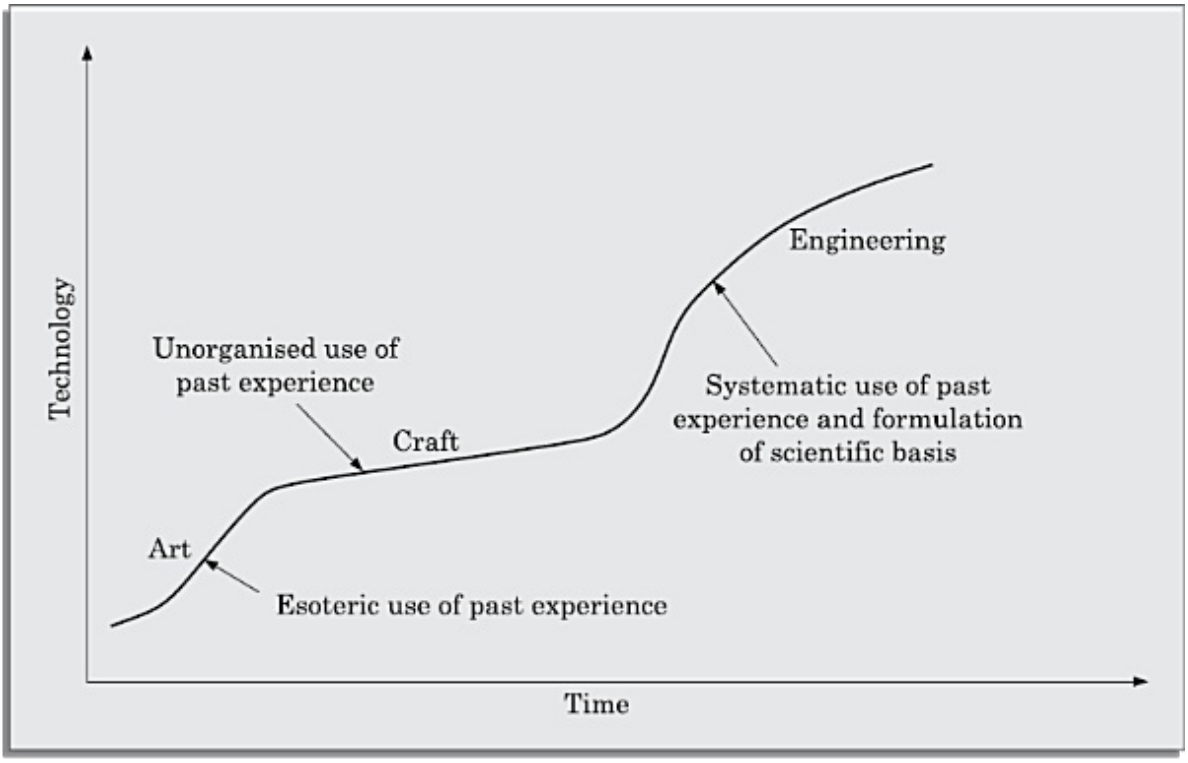

Figure 1.1: Evolution of technology with time.

Software engineering principles are now being widely used in industry, and new principles are still continuing to emerge at a very rapid rate—making this discipline highly dynamic. In spite of its wide acceptance, critics point out that many of the methodologies and guidelines provided by the software engineering discipline lack scientific basis, are subjective, and often inadequate. Yet, there is no denying the fact that adopting software engineering techniques facilitates development of high quality software in a cost-effective and timely manner. Software engineering practices have proven to be indispensable to the development of large software products—though exploratory styles are often used successfully to develop small programs such as those written by students as classroom assignments.

1.1.3 A Solution to the Software Crisis

At present, software engineering appears to be among the few options that are available to tackle the present software crisis. But, what exactly is the present software crisis? What are its symptoms, causes, and possible solutions? To understand the present software crisis, consider the following facts. The expenses that organisations all over the world are incurring on software purchases as compared to the expenses incurred on hardware purchases have been showing an worrying trend over the years (see Figure 1.2). As can be seen in the

figure, organisations are spending increasingly larger portions of their budget on software as compared to that on hardware. Among all the symptoms of the present software crisis, the trend of increasing software costs is probably the most vexing.

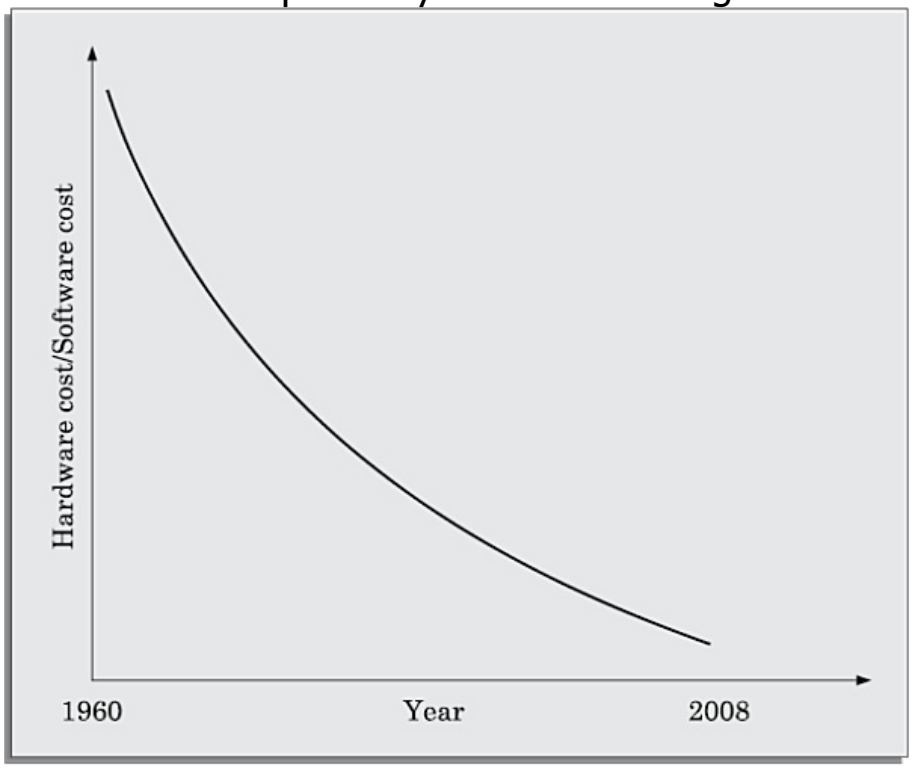

Figure 1.2: Relative changes of hardware and software costs over time.

Not only are the software products becoming progressively more expensive than hardware, but they also present a host of other problems to the customers—software products are difficult to alter, debug, and enhance; use resources non-optimally; often fail to meet the user requirements; are far from being reliable; frequently crash; and are often delivered late.

At present, many organisations are actually spending much more on software than on hardware. If this trend continues, we might soon have a rather amusing scenario. Not long ago, when you bought any hardware product, the essential software that ran on it came free with it. But, unless some sort of revolution happens, in not very distant future, hardware prices would become insignificant compared to software prices—when you buy any software product the hardware on which the software runs would come free with the software!!!

The symptoms of software crisis are not hard to observe. But, what are the factors that have contributed to the present software crisis? Apparently, there are many factors, the important ones being—rapidly increasing problem size, lack of adequate training in software engineering techniques, increasing skill

shortage, and low productivity improvements. What is the remedy? It is believed that a satisfactory solution to the present software crisis can possibly come from a spread of software engineering practices among the developers, coupled with further advancements to the software engineering discipline itself.

With this brief discussion on the evolution and impact of the discipline of software engineering, we now examine some basic concepts pertaining to the different types of software development projects that are undertaken by software companies.

1.2 SOFTWARE DEVELOPMENT PROJECTS

Before discussing about the various types of development projects that are being undertaken by software development companies, let us first understand the important ways in which professional software differs from toy software such as those written by a student in his first programming assignment.

Programs *versus* Products

Many toy software are being developed by individuals such as students for their classroom assignments and hobbyists for their personal use. These are usually small in size and support limited functionalities. Further, the author of a program is usually the sole user of the software and himself maintains the code. These toy software therefore usually lack good user-interface and proper documentation. Besides these may have poor maintainability, efficiency, and reliability. Since these toy software do not have any supporting documents such as users' manual, maintenance manual, design document, test documents, etc., we call these toy software as *programs*.

In contrast, professional software usually have multiple users and, therefore, have good user-interface, proper users' manuals, and good documentation support. Since, a software product has a large number of users, it is systematically designed, carefully implemented, and thoroughly tested. In addition, a professionally written software usually consists not only of the program code but also of all associated documents such as requirements specification document, design document, test document, users' manuals, etc. A further difference is that professional software are often too large and complex to be developed by any single individual. It is usually developed by a group of developers working in a team.

A professional software is developed by a group of software developers working together in a team. It is therefore necessary for them to use some systematic development methodology. Otherwise, they would find it very difficult to interface and understand each other's work, and produce a coherent set of documents.

Even though software engineering principles are primarily intended for use in development of professional software, many results of software engineering can effectively be used for development of small programs as well. However, when developing small programs for personal use, rigid adherence to software engineering principles is often not worthwhile. An ant can be killed using a gun, but it would be ridiculously inefficient and inappropriate. CAR Hoare [1994] observed that rigorously using software engineering principles to develop toy programs is very much like employing civil and architectural engineering principles to build sand castles for children to play.

1.2.1 Types of Software Development Projects

A software development company is typically structured into a large number of teams that handle various types of software development projects. These software development projects concern the development of either software product or some software service. In the following subsections, we distinguish between these two types of software development projects.

Software products

We all know of a variety of software such as Microsoft's Windows and the Office suite, Oracle DBMS, software accompanying a camcorder or a laser printer, etc. These software are available off-the-shelf for purchase and are used by a diverse range of customers. These are called *generic software products* since many users essentially use the same software. These can be purchased off-the-shelf by the customers. When a software development company wishes to develop a generic product, it first determines the features or functionalities that would be useful to a large cross section of users. Based on these, the development team draws up the product specification on its own. Of course, it may base its design discretion on feedbacks collected from a large number of users. Typically, each software product is targetted to some market segment (set of users). Many companies find it

advantageous to develop *product lines* that target slightly different market segments based on variations of essentially the same software. For example, Microsoft targets desktops and laptops through its *Windows* 8 operating system, while it targets high-end mobile handsets through its *Windows mobile* operating system, and targets servers through its *Windows server* operating system.

Software services

A software service usually involves either development of a *customised software* or development of some specific part of a software in an outsourced mode. A *customised software* is developed according to the specification drawn up by one or at most a few customers. These need to be developed in a short time frame (typically a couple of months), and at the same time the development cost must be low. Usually, a developing company develops customised software by tailoring some of its existing software. For example, when an academic institution wishes to have a software that would automate its important activities such as student registration, grading, and fee collection; companies would normally develop such a software as a customised product. This means that for developing a customised software, the developing company would normally tailor one of its existing software products that it might have developed in the past for some other academic institution.

In a customised software development project, a large part of the software is reused from the code of related software that the company might have already developed. Usually, only a small part of the software that is specific to some client is developed. For example, suppose a software development organisation has developed an academic automation software that automates the student registration, grading, Establishment, hostel and other aspects of an academic institution. When a new educational institution requests for developing a software for automation of its activities, a large part of the existing software would be reused. However, a small part of the existing code may be modified to take into account small variations in the required features. For example, a software might have been developed for an academic institute that offers only regular residential programs, the educational institute that has now requested for a software to automate its activities also offers a distance mode post graduate program where the teaching and sessional evaluations are done by the local centres.

Another type of software service is *outsourced software*. Sometimes, it can

make good commercial sense for a company developing a large project to outsource some parts of its development work to other companies. The reasons behind such a decision may be many. For example, a company might consider the outsourcing option, if it feels that it does not have sufficient expertise to develop some specific parts of the software; or if it determines that some parts can be developed cost-effectively by another company. Since an outsourced project is a small part of some larger project, outsourced projects are usually small in size and need to be completed within a few months or a few weeks of time.

The types of development projects that are being undertaken by a company can have an impact on its profitability. For example, a company that has developed a generic software product usually gets an uninterrupted stream of revenue that is spread over several years. However, this entails substantial upfront investment in developing the software and any return on this investment is subject to the risk of customer acceptance. On the other hand, outsourced projects are usually less risky, but fetch only one time revenue to the developing company.

1.2.2 Software Projects Being Undertaken by Indian Companies

Indian software companies have excelled in executing software services projects and have made a name for themselves all over the world. Of late, the Indian companies have slowly started to focus on product development as well. Can you recall the names of a few software products developed by Indian software companies? Let us try to hypothesise the reason for this situation. Generic product development entails certain amount of business risk. A company needs to invest upfront and there is substantial risks concerning whether the investments would turn profitable. Possibly, the Indian companies were risk averse.

Till recently, the world-wide sales revenue of software products and services were evenly matched. But, of late the services segment has been growing at a faster pace due to the advent of application service provisioning and cloud computing. We discuss these issues in Chapter 15.

1.3 EXPLORATORY STYLE OF SOFTWARE DEVELOPMENT

We have already discussed that the *exploratory program development style* refers to an informal development style where the programmer makes use of his own intuition to develop a program rather than making use of

the systematic body of knowledge categorized under the software engineering discipline. The exploratory development style gives complete freedom to the programmer to choose the activities using which to develop software. Though the exploratory style imposes no rules a typical development starts after an initial briefing from the customer. Based on this briefing, the developers start coding to develop a working program. The software is tested and the bugs found are fixed. This cycle of testing and bug fixing continues till the software works satisfactorily for the customer. A schematic of this work sequence in a build and fix style has been shown graphically in Figure 1.3. Observe that coding starts after an initial customer briefing about what is required. After the program development is complete, a test and fix cycle continues till the program becomes acceptable to the customer.

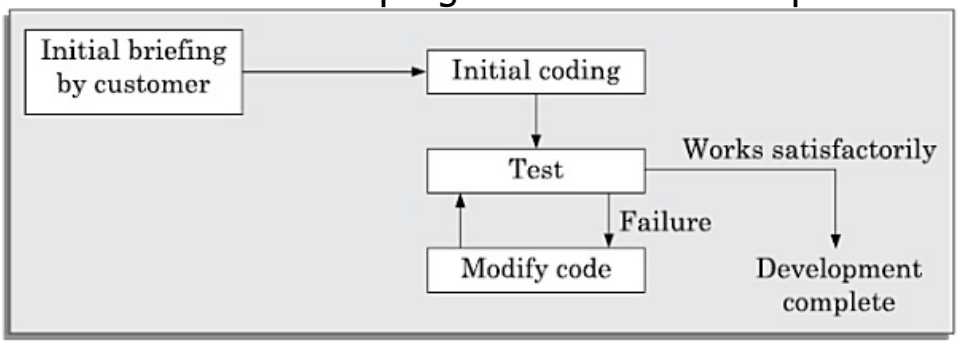

Figure 1.3: Exploratory program development.

An exploratory development style can be successful when used for developing very small programs, and not for professional software. We had examined this issue with the help of the petty contractor analogy. Now let us examine this issue more carefully.

What is wrong with the exploratory style of software development?

Though the exploratory software development style is intuitively obvious, no software team can remain competitive if it uses this style of software development. Let us investigate the reasons behind this. In an exploratory development scenario, let us examine how do the effort and time required to develop a professional software increases with the increase in program size. Let us first consider that exploratory style is being used to develop a professional software. The increase in development effort and time with problem size has been indicated in Figure 1.4. Observe the thick line plot that represents the case in which the exploratory style i s used to develop a program. It can be seen that as the program size increases, the required effort and time increases almost exponentially. For large problems, it would

take too long and cost too much to be practically meaningful to develop the program using the exploratory style of development. The exploratory development approach is said to break down after the size of the program to be developed increases beyond certain value. For example, using the exploratory style, you may easily solve a problem that requires writing only 1000 or 2000 lines of source code. But, if you are asked to solve a problem that would require writing one million lines of source code, you may never be able to complete it using the exploratory style; irrespective of the amount time or effort you might invest to solve it. Now observe the thin solid line plot in Figure 1.4 which represents the case when development is carried out using software engineering principles. In this case, it becomes possible to solve a problem with effort and time that is almost linear in program size. On the other hand, if programs could be written automatically by machines, then the increase in effort and time with size would be even closer to a linear (dotted line plot) increase with size.

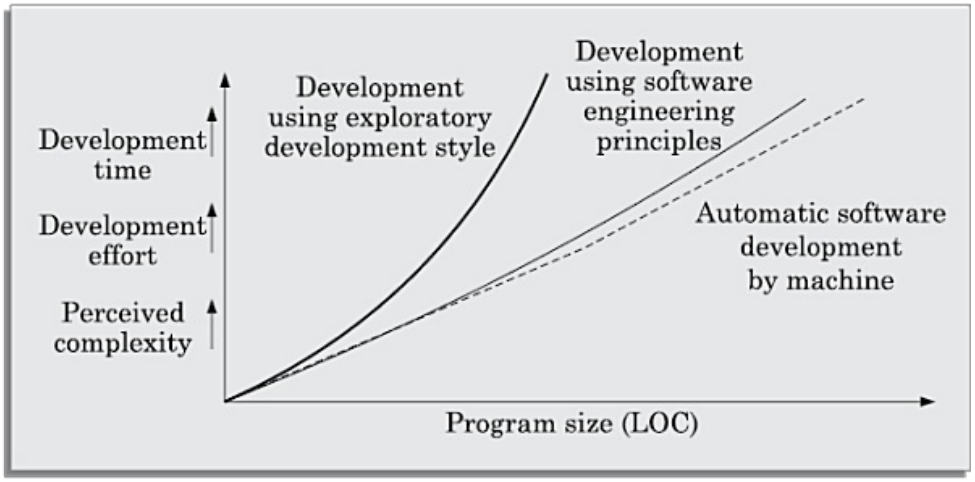

Figure 1.4: Increase in development time and effort with problem size.

Now let us try to understand why does the effort required to develop a program grow exponentially with program size when the exploratory style is used and then this approach to develop a program completely breaks down when the program size becomes large? To get an insight into the answer to this question, we need to have some knowledge of the human cognitive limitations (see the discussion on human psychology in subsection 1.3.1). As we shall see, the perceived (or psychological) complexity of a problem grows exponentially with its size. Please note that the perceived complexity of a problem is not related to the time or space complexity issues with which you are likely to be familiar with from a basic course on algorithms.

The psychological or perceived complexity of a problem concerns the difficulty level

experienced by a programmer while solving the problem using the exploratory development style.

Even if the exploratory style causes the perceived difficulty of a problem to grow exponentially due to human cognitive limitations, how do the software engineering principles help to contain this exponential rise in complexity with problem size and hold it down to almost a linear increase? We will discuss in subsection 1.3.2 that software engineering principle help achieve this by profusely making use of the abstraction and decomposition techniques to overcome the human cognitive limitations. You may still wonder that when software engineering principles are used, why does the curve not become completely linear? The answer is that i t is very difficult to apply the decomposition and abstraction principles to completely overcome the problem complexity.

Summary of the shortcomings of the exploratory style of software development:

We briefly summarise the important shortcomings of using the exploratory development style to develop a professional software:

- The foremost difficulty is the exponential growth of development time and effort with problem size and large-sized software becomes almost impossible using this style of development.
- The exploratory style usually results in unmaintainable code. The reason for this is that any code developed without proper design would result in highly unstructured and poor quality code.
- It becomes very difficult to use the exploratory style in a team development environment. In the exploratory style, the development work is undertaken without any proper design and documentation. Therefore it becomes very difficult to meaningfully partition the work among a set of developers who can work concurrently. On the other hand, team development is indispensable for developing modern software—most software mandate huge development efforts, necessitating team effort for developing these. Besides poor quality code, lack of proper documentation makes any later maintenance of the code very difficult.

1.3.1 Perceived Problem Complexity: An Interpretation Based on

Human Cognition Mechanism

The rapid increase of the perceived complexity of a problem with increase in problem size can be explained from an interpretation of the human cognition mechanism. A simple understanding of the human cognitive mechanism would also give us an insight into why the exploratory style of development leads to an undue increase in the time and effort required to develop a programming solution. It can also explain why it becomes practically infeasible to solve problems larger than a certain size while using an exploratory style; whereas using software engineering principles, the required effort grows almost linearly with size (as indicated by the thin solid line in Figure 1.4).

Psychologists say that the human memory can be thought to consist of two distinct parts[Miller 56]: short-term and long-term memories. A schematic representation of these two types of memories and their roles in human cognition mechanism has been shown in Figure 1.5. In Figure 1.5, the block labelled sensory organs represents the five human senses sight, hearing, touch, smell, and taste. The block labelled actuator represents neuromotor organs such as hand, finger, feet, etc. We now elaborate this human cognition model in the following subsection.

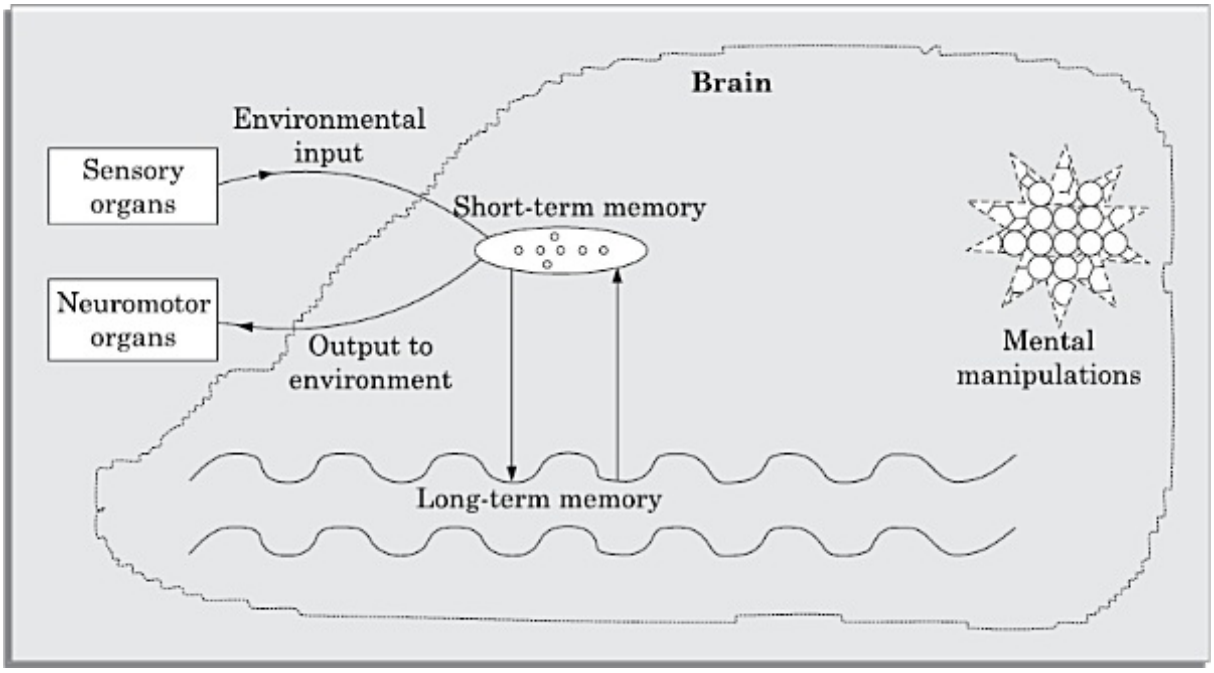

Figure 1.5: Human cognition mechanism model.

Short-term memory: The short-term memory, as the name itself suggests, can store information for a short while—usually up to a few seconds, and at most for a few minutes. The short-term memory is also sometimes referred to as

the *working memory*. The information stored in the short-term memory is immediately accessible for processing by the brain. The short-term memory of an average person can store up to seven items; but in extreme cases it can vary anywhere from five to nine items (7 ± 2). As shown in Figure 1.5, the short-term memory participates in all interactions of the human mind with its environment.

It should be clear that the short-term memory plays a very crucial part in the human cognition mechanism. All information collected through the sensory organs are first stored in the short-term memory. The short-term memory is also used by the brain to drive the neuromotor organs. The mental manipulation unit also gets its inputs from the short-term memory and stores back any output it produces. Further, information retrieved from the long-term memory first gets stored in the short-term memory. For example, if you are asked the question: "If it is 10AM now, how many hours are remaining today?" First, 10AM would be stored in the short-term memory. Next, the information that a day is 24 hours long would be fetched from the long-term memory into the short-term memory. The mental manipulation unit would compute the difference (24–10), and 14 hours would get stored in the short-term memory. As you can notice, this model is very similar to the organisation of a computer in terms of cache, main memory, and processor.

An item stored in the short-term memory can get lost either due to decay with time or displacement by newer information. This restricts the duration for which an item is stored in the short-term memory to few tens of seconds. However, a n item can be retained longer in the short-term memory by recycling. That is, when we repeat or refresh a n item consciously, we can remember it for a much longer duration. Certain information stored in the short-term memory, under certain circumstances gets stored in the long-term memory.

Long-term memory: Unlike the short-term memory, the size of the long-term memory is not known to have a definite upper bound. The size of the long-term memory can vary from several million items to several billion items, largely depending on how actively a person exercises his mental faculty. An item once stored in the long-term memory, is usually retained for several years. But, how do items get stored in the long-term memory? Items present in the short-term memory can get stored in the long-term memory either through large number of refreshments (repetitions) or by forming links with already existing items in the long-term memory. For example, you possibly remember your own telephone number because you might have repeated

(refreshed) it for a large number of times in your short-term memory. Let us now take an example of a situation where you may form links to existing items in the long- term memory to remember certain information. Suppose you want to remember the 10 digit mobile number 9433795369. To remember it by rote may be intimidating. But, suppose you consider the number as split into 9433 7953 69 and notice that 94 is the code for BSNL, 33 is the code for Kolkata, suppose 79 is your year of birth, and 53 is your roll number, and the rest of the two numbers are each one less than the corresponding digits of the previous number; you have effectively established links with already stored items, making it easier to remember the number.

Item: We have so far only mentioned the number of items that the long-term and the short-term memories can store. But, what exactly is an item? An *item* is any set of related information. According to this definition, a character such as *a* or a digit such as '5' can each be considered as an item. A word, a sentence, a story, or even a picture can each be a single item. Each item normally occupies one place in memory. The definition of an item as any set of related information implies that when you are able to establish some simple relationship between several different items, the information that should normally occupy several places can be stored using only one place in the memory. This phenomenon of forming one item from several items is referred to as *chunking* by psychologists. For example, if you are given the binary number 110010101001—it may prove very hard for you to understand and remember. But, the octal form of the number 6251 (i.e., the representation (110)(010)(101)(001)) may be much easier to understand and remember since we have managed to create chunks of three items each.

Evidence of short-term memory: Evidences of short-term memory manifest themselves in many of our day-to-day experiences. As an example of the short-term memory, consider the following situation. Suppose, you look up a number from the telephone directory and start dialling it. If you find the number to be busy, you would dial the number again after a few seconds—in this case, you would be able to do so almost effortlessly without having to look up the directory. But, after several hours or days since you dialled the number last, you may not remember the number at all, and would need to consult the directory again.

The magical number 7: Miller called the number seven as the *magical number* [Miller 56] since if a person deals with seven or less number of unrelated information at a time these would be easily accommodated in the short-term

memory. So, he can easily understand it. As the number of items that one has to deal with increases beyond seven, it becomes exceedingly difficult to understand it. This observation can easily be extended to writing programs.

> When the number of details (or variables) that one has to track to solve a problem increases beyond seven, it exceeds the capacity of the short-term memory and it becomes exceedingly more difficult for a human mind to grasp the problem.

A small program having just a few variables is within the easy grasp of an individual. As the number of independent variables in the program increases, it quickly exceeds the grasping power of an individual and would require an unduly large effort to master the problem. This outlines a possible reason behind the exponential nature of the effort-size plot (thick line) shown in Figure 1.4. Please note that the situation depicted in Figure 1.4 arises mostly due to the human cognitive limitations. Instead of a human, if a machine could be writing (generating) a program, the slope of the curve would be linear, as the cache size (short-term memory) of a computer is quite large. But, why does the effort-size curve become almost linear when software engineering principles are deployed? This is because software engineering principles extensively use the techniques that are designed specifically to overcome the human cognitive limitations. We discuss this issue in the next subsection.

1.3.2 Principles Deployed by Software Engineering to Overcome Human Cognitive Limitations

We shall see throughout this book that a central theme of most of software engineering principles is the use of techniques to effectively tackle the problems that arise due to human cognitive limitations.

> Two important principles that are deployed by software engineering to overcome the problems arising due to human cognitive limitations are—abstraction and decomposition.

In the following subsections, with the help of Figure 1.6(a) and (b), we explain the essence of these two important principles and how they help to overcome the human cognitive limitations. In the rest of this book, we shall time and again encounter the use of these two fundamental principles in various forms and flavours in the different software development activities. A thorough understanding of these two principles is therefore needed.

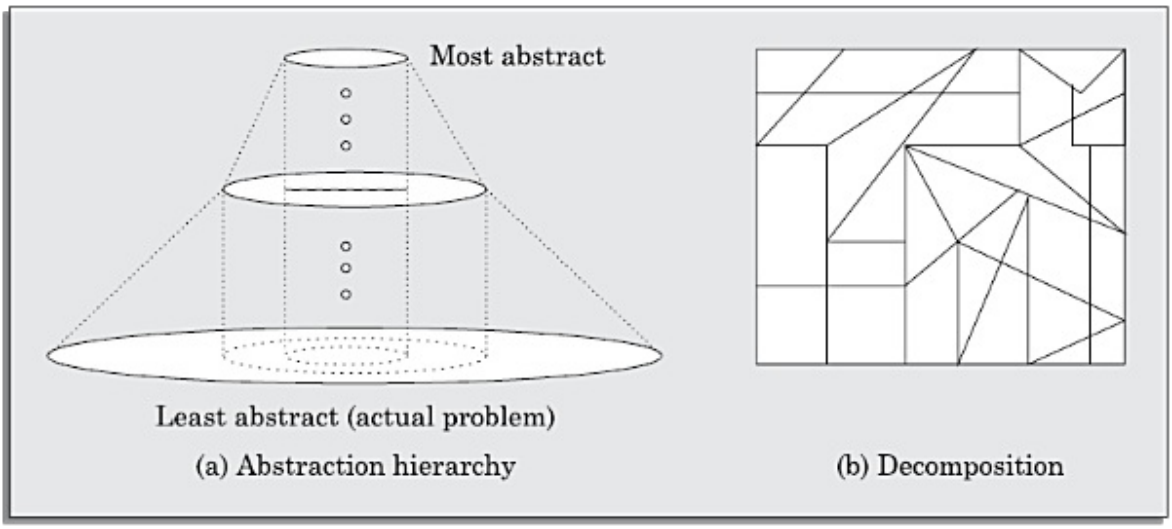

Figure 1.6: Schematic representation.

Abstraction

Abstraction refers to construction of a simpler version of a problem by ignoring the details. The principle of constructing an abstraction is popularly known as *modelling* (or *model construction*).

> Abstraction is the simplification of a problem by focusing on only one aspect of the problem while omitting all other aspects.

When using the principle of abstraction to understand a complex problem, we focus our attention on only one or two specific aspects of the problem and ignore the rest. Whenever we omit some details of a problem to construct an abstraction, we construct a *model* of the problem. In every day life, we use the principle of abstraction frequently to understand a problem or to assess a situation. Consider the following two examples.

- Suppose you are asked to develop an overall understanding of some country. No one in his right mind would start this task by meeting all the citizens of the country, visiting every house, and examining every tree of the country, etc. You would probably take the help of several types of abstractions to do this. You would possibly start by referring to and understanding various types of maps for that country. A map, in fact, is an abstract representation of a country. It ignores detailed information such as the specific persons who inhabit it, houses, schools, play grounds, trees, etc. Again, there are two important types of maps—physical and political maps. A *physical map* shows the physical features of an area; such as mountains, lakes, rivers, coastlines, and so

on. On the other hand, the *political map* shows states, capitals, and national boundaries, etc. The physical map is an abstract model of the country and ignores the state and district boundaries. The political map, on the other hand, is another abstraction of the country that ignores the physical characteristics such as elevation of lands, vegetation, etc. It can be seen that, for the same object (e.g. country), several abstractions are possible. In each abstraction, some aspects of the object is ignored. We understand a problem by abstracting out different aspects of a problem (constructing different types of models) and understanding them. It is not very difficult to realise that proper use of the principle of abstraction can be a very effective help to master even intimidating problems.

- Consider the following situation. Suppose you are asked to develop an understanding of all the living beings inhabiting the earth. If you use the naive approach, you would start taking up one living being after another who inhabit the earth and start understanding them. Even after putting in tremendous effort, you would make little progress and left confused since there are billions of living things on earth and the information would be just too much for any one to handle. Instead, what can be done is to build and understand an abstraction hierarchy of all living beings as shown in Figure 1.7. At the top level, we understand that there are essentially three fundamentally different types of living beings—plants, animals, and fungi. Slowly more details are added about each type at each successive level, until we reach the level of the different species at the leaf level of the abstraction tree.

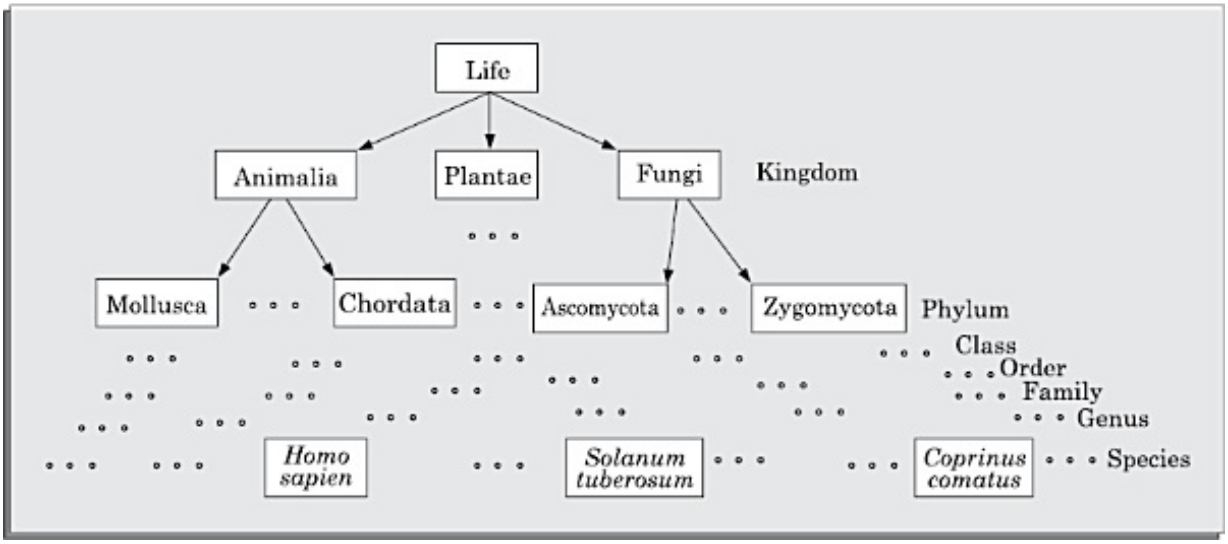

Figure 1.7: An abstraction hierarchy classifying living organisms.

A single level of abstraction can be sufficient for rather simple problems. However, more complex problems would need to be modelled as a hierarchy of abstractions. A schematic representation of an abstraction hierarchy has been shown in Figure 1.6(a). The most abstract representation would have only a few items and would be the easiest to understand. After one understands the simplest representation, one would try to understand the next level of abstraction where at most five or seven new information are added and so on until the lowest level is understood. By the time, one reaches the lowest level, he would have mastered the entire problem.

Decomposition

Decomposition is another important principle that is available in the repertoire of a software engineer to handle problem complexity. This principle is profusely made use by several software engineering techniques to contain the exponential growth of the perceived problem complexity. The decomposition principle is popularly known as the *divide and conquer* principle.

> The decomposition principle advocates decomposing the problem into many small independent parts. The small parts are then taken up one by one and solved separately. The idea is that each small part would be easy to grasp and understand and can be easily solved. The full problem is solved when all the parts are solved.

A popular way to demonstrate the decomposition principle is by trying to break a large bunch of sticks tied together and then breaking them individually. Figure 1.6(b) shows the decomposition of a large problem into many small parts. However, it is very important to understand that any arbitrary decomposition of a problem into small parts would not help. The different parts after decomposition should be more or less independent of each other. That is, to solve one part you should not have to refer and understand other parts. If to solve one part you would have to understand other parts, then this would boil down to understanding all the parts together. This would effectively reduce the problem to the original problem before decomposition (the case when all the sticks tied together). Therefore, it is not sufficient to just decompose the problem in any way, but the decomposition should be such that the different decomposed parts must be more or less independent of each other.

As an example of a use of the principle of decomposition, consider the following. You would understand a book better when the contents are decomposed (organised) into more or less independent chapters. That is,

each chapter focuses on a separate topic, rather than when the book mixes up all topics together throughout all the pages. Similarly, each chapter should be decomposed into sections such that each section discusses a different issue. Each section should be decomposed into subsections and so on. If various subsections are nearly independent of each other, the subsections can be understood one by one rather than keeping on cross referencing to various subsections across the book to understand one.

Why study software engineering?

Let us examine the skills that you could acquire from a study of the software engineering principles. The following two are possibly the most important skill you could be acquiring after completing a study of software engineering:

- The skill to participate in development of large software. You can meaningfully participate in a team effort to develop a large software only after learning the systematic techniques that are being used in the industry.
- You would learn how to effectively handle complexity in a software development problem. In particular, you would learn how to apply the principles of abstraction and decomposition to handle complexity during various stages in software development such as specification, design, construction, and testing.

Besides the above two important skills, you would also be learning the techniques of software requirements specification user interface development, quality assurance, testing, project management, maintenance, etc.

As we had already mentioned, small programs can also be written without using software engineering principles. However even if you intend to write small programs, the software engineering principles could help you to achieve higher productivity and at the same time enable you to produce better quality programs.

1.4 EMERGENCE OF SOFTWARE ENGINEERING

We have already pointed out that software engineering techniques have evolved over many years in the past. This evolution is the result of a series of innovations and accumulation of experience about writing

good quality programs. Since these innovations and programming experiences are too numerous, let us briefly examine only a few of these innovations and programming experiences which have contributed to the development of the software engineering discipline.

1.4.1 Early Computer Programming

Early commercial computers were very slow and too elementary as compared to today's standards. Even simple processing tasks took considerable computation time on those computers. No wonder that programs at that time were very small in size and lacked sophistication. Those programs were usually written in assembly languages. Program lengths were typically limited to about a few hundreds of lines of monolithic assembly code. Every programmer developed his own individualistic style of writing programs according to his intuition and used this style *ad hoc* while writing different programs. In simple words, programmers wrote programs without formulating any proper solution strategy, plan, or design a jump to the terminal and start coding immediately on hearing out the problem. They then went on fixing any problems that they observed until they had a program that worked reasonably well. We have already designated this style of programming as the *build and fix* (or the *exploratory programming*) style.

1.4.2 High-level Language Programming

Computers became faster with the introduction of the semiconductor technology in the early 1960s. Faster semiconductor transistors replaced the prevalent vacuum tube-based circuits in a computer. With the availability of more powerful computers, it became possible to solve larger and more complex problems. At this time, high-level languages such as FORTRAN, ALGOL, and COBOL were introduced. This considerably reduced the effort required to develop software and helped programmers to write larger programs (why?). Writing each high-level programming construct in effect enables the programmer to write several machine instructions. Also, the machine details (registers, flags, etc.) are abstracted from the programmer. However, programmers were still using the exploratory style of software development. Typical programs were limited to sizes of around a few thousands of lines of source code.

1.4.3 Control Flow-based Design

As the size and complexity of programs kept on increasing, the exploratory programming style proved to be insufficient. Programmers found it increasingly difficult not only to write cost-effective and correct programs, but also to understand and maintain programs written by others. To cope up with this problem, experienced programmers advised other programmers to pay particular attention to the design of a program's *control flow structure*.

A program's control flow structure indicates the sequence in which the program's instructions are executed.

In order to help develop programs having good control flow structures, the *flow charting technique* was developed. Even today, the flow charting technique is being used to represent and design algorithms; though the popularity of flow charting represent and design programs has want to a great extent due to the emergence of more advanced techniques.

Figure 1.8 illustrates two alternate ways of writing program code for the same problem. The flow chart representations for the two program segments of Figure 1.8 are drawn in Figure 1.9. Observe that the control flow structure of the program segment in Figure 1.9(b) is much more simpler than that of Figure 1.9(a). By examining the code, it can be seen that Figure 1.9(a) is much harder to understand as compared to Figure 1.9(b). This example corroborates the fact that if the flow chart representation is simple, then the corresponding code should be simple. You can draw the flow chart representations of several other problems to convince yourself that a program with complex flow chart representation is indeed more difficult to understand and maintain.

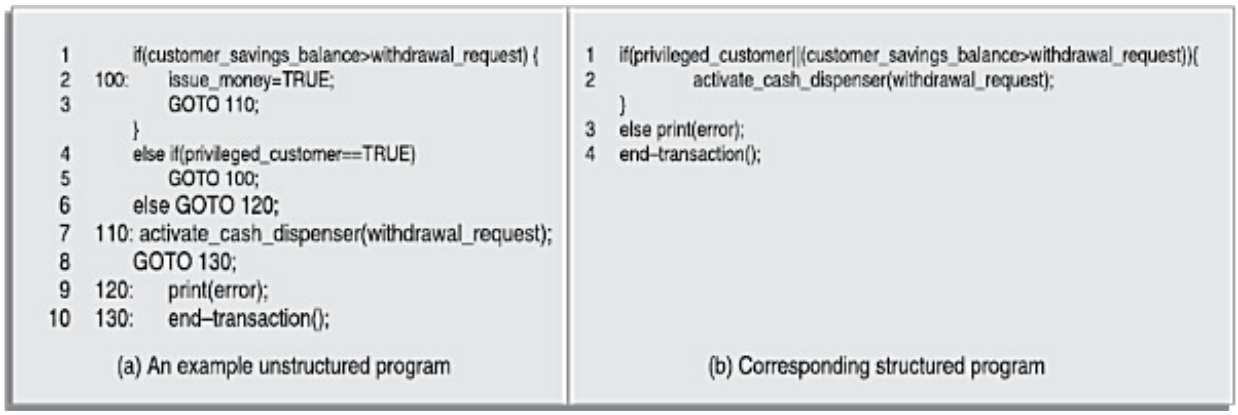

Figure 1.8: An example of (a) Unstructured program (b) Corresponding structured program.

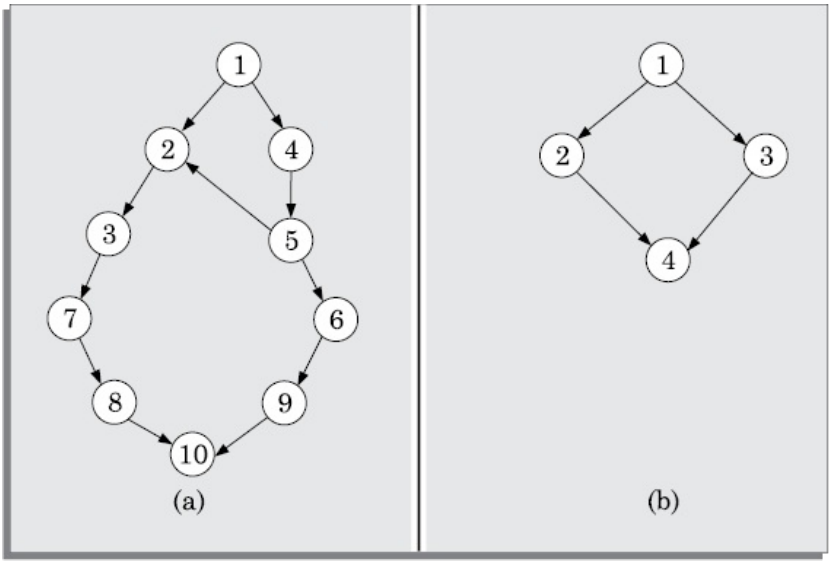

Figure 1.9: Control flow graphs of the programs of Figures 1.8(a) and (b).

Let us now try to understand why a program having good control flow structure would be easier to develop and understand. In other words, let us understand why a program with a complex flow chart representation is difficult to understand? The main reason behind this situation is that normally one understands a program by mentally tracing its execution sequence (i.e. statement sequences) to understand how the output is produced from the input values. That is, we can start from a statement producing an output, and trace back the statements in the program and understand how they produce the output by transforming the input data. Alternatively, we may start with the input data and check by running through the program how each statement processes (transforms) the input data until the output is produced. For example, for the program of Fig 1.9(a) you would have to understand the execution of the program along the paths 1-2-3-7-8-10, 1-4-5-6-9-10, and 1-4-5-2-3-7-8-10. A program having a messy control flo w (i.e. flow chart) structure, would have a large number of execution paths (see Figure 1.10). Consequently, it would become extremely difficult to determine all the execution paths, and tracing the execution sequence along all the paths trying to understand them can be nightmarish. It is therefore evident that a program having a messy flow chart representation would indeed be difficult to understand and debug.

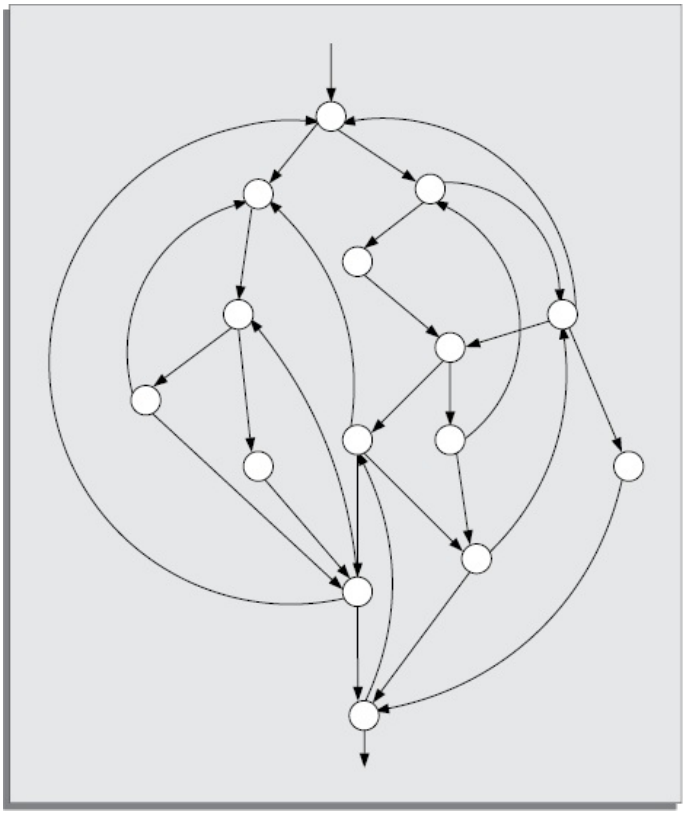

Figure 1.10: CFG of a program having too many GO TO statements.

Are GO TO statements the culprits?

In a landmark paper, Dijkstra [1968] published his (now famous) article "GO TO Statements Considered Harmful". He pointed out that unbridled use of GO TO statements is the main culprit in making the control structure of a program messy. To understand his argument, examine Figure 1.10 which shows the flow chart representation of a program in which the programmer has used rather too many GO TO statements. GO TO statements alter the flow of control arbitrarily, resulting in too many paths. But, then why does use of too many GO TO statements makes a program hard to understand?

A programmer trying to understand a program would have to mentally trace and understand the processing that take place along all the paths of the program making program understanding and debugging extremely complicated.

Soon it became widely accepted that good programs should have very simple control structures. It is possible to distinguish good programs from bad programs by just visually examining their flow chart representations. The use of flow charts to design good control flow structures of programs became wide spread.

Structured programming—a logical extension

The need to restrict the use of GO TO statements was recognised by everybody. However, many programmers were still using assembly languages. JUMP instructions are frequently used for program branching in assembly languages. Therefore, programmers with assembly language programming background considered the use of GO TO statements in programs inevitable. However, it was conclusively proved by Bohm and Jacopini that only three programming constructs—sequence, selection, and iteration—were sufficient to express any programming logic. This was an important result—it is considered important even today. An example of a sequence statement is an assignment statement of the form a=b;. Examples of selection and iteration statements are the if-then-else and the do-while statements respectively. Gradually, everyone accepted that it is indeed possible to solve any programming problem without using GO TO statements and that indiscriminate use of GO TO statements should be avoided. This formed the basis of the structured programming methodology.

> A program is called structured when it uses only the sequence, selection, and iteration types of constructs and is modular.

Structured programs avoid unstructured control flows by restricting the use of GO TO statements. Structured programming is facilitated, if the programming language being used supports single-entry, single-exit program constructs such as if-then-else, do-while, etc. Thus, an important feature of structured programs is the design of good control structures. An example illustrating this key difference between structured and unstructured programs is shown in Figure 1.8. The program in Figure 1.8(a) makes use of too many GO TO statements, whereas the program in Figure 1.8(b) makes use of none. The flow chart of the program making use of GO TO statements is obviously much more complex as can be seen in Figure 1.9.

Besides the control structure aspects, the term *structured program* is being used to denote a couple of other program features as well. A structured program should be modular. A modular program is one which is decomposed into a set of modules[1] such that the modules should have low interdependency among each other. We discuss the concept of modular programs in Chapter 5.

But, what are the main advantages of writing structured programs compared to the unstructured ones? Research experiences have shown that

programmers commit less number of errors while using structured if-then-else and do-while statements than when using test-and-branch code constructs. Besides being less error-prone, structured programs are normally more readable, easier to maintain, and require less effort to develop compared to unstructured programs. The virtues of structured programming became widely accepted and the structured programming concepts are being used even today. However, violations to the structured programming feature is usually permitted in certain specific programming situations, such as exception handling, etc.

Very soon several languages such as PASCAL, MODULA, C, etc., became available which were specifically designed to support structured programming. These programming languages facilitated writing modular programs and programs having good control structures. Therefore, messy control structure was no longer a big problem. So, the focus shifted from designing good control structures to designing good data structures for programs.

1.4.4 Data Structure-oriented Design

Computers became even more powerful with the advent of *integrated circuits* (ICs) in the early seventies. These could now be used to solve more complex problems. Software developers were tasked to develop larger and more complicated software. which often required writing in excess of several tens of thousands of lines of source code. The control flow-based program development techniques could not be used satisfactorily any more to write those programs, and more effective program development techniques were needed.

It was soon discovered that while developing a program, it is much more important to pay attention to the design of the important data structures of the program than to the design of its control structure. Design techniques based on this principle are called *data structure- oriented* design techniques.

> Using data structure-oriented design techniques, first a program's data structures are designed. The code structure is designed based on the data structure.

In the next step, the program design is derived from the data structure. An example of a data structure-oriented design technique is the Jackson's Structured Programming (JSP) technique developed by Michael Jackson [1975]. In JSP methodology, a program's data structure is first designed using the notations for sequence, selection, and iteration. The JSP methodology

provides an interesting technique to derive the program structure from its data structure representation. Several other data structure-based design techniques were also developed. Some of these techniques became very popular and were extensively used. Another technique that needs special mention is the Warnier-Orr Methodology [1977, 1981]. However, we will not discuss these techniques in this text because now-a-days these techniques are rarely used in the industry and have been replaced by the data flow-based and the object-oriented techniques.

1.4.5 Data Flow-oriented Design

As computers became still faster and more powerful with the introduction of *very large scale integrated* (VLSI) Circuits and some new architectural concepts, more complex and sophisticated software were needed to solve further challenging problems. Therefore, software developers looked out for more effective techniques for designing software and soon *data flow-oriented techniques* were proposed.

> The data flow-oriented techniques advocate that the major data items handled by a system must be identified and the processing required on these data items to produce the desired outputs should be determined.

The functions (also called as *processes*) and the data items that are exchanged between the different functions are represented in a diagram known as a *data flow diagram* (DFD). The program structure can be designed from the DFD representation of the problem.

DFDs: A crucial program representation for procedural program design

DFD has proven to be a generic technique which is being used to model all types of systems, and not just software systems. For example, Figure 1.11 shows the data-flow representation of an automated car assembly plant. If you have never visited an automated car assembly plant, a brief description of an automated car assembly plant would be necessary. In an automated car assembly plant, there are several processing stations (also called *workstations*) which are located along side of a conveyor belt (also called an *assembly line*). Each workstation is specialised to do jobs such as fitting of wheels, fitting the engine, spray painting the car, etc. As the partially assembled program moves along the assembly line, different workstations perform their respective jobs on the partially assembled software. Each circle in the DFD

model of Figure 1.11 represents a workstation (called a *process* or *bubble*). Each workstation consumes certain input items and produces certain output items. As a car under assembly arrives at a workstation, it fetches the necessary items to be fitted from the corresponding stores (represented by two parallel horizontal lines), and as soon as the fitting work is complete passes on to the next workstation. It is easy to understand the DFD model of the car assembly plant shown in Figure 1.11 even without knowing anything regarding DFDs. In this regard, we can say that a major advantage of the DFDs is their simplicity. In Chapter 6, we shall study how to construct the DFD model of a software system. Once you develop the DFD model of a problem, data flow-oriented design techniques provide a rather straight forward methodology to transform the DFD representation of a problem into an appropriate software design. We shall study the data flow-based design techniques in Chapter 6.

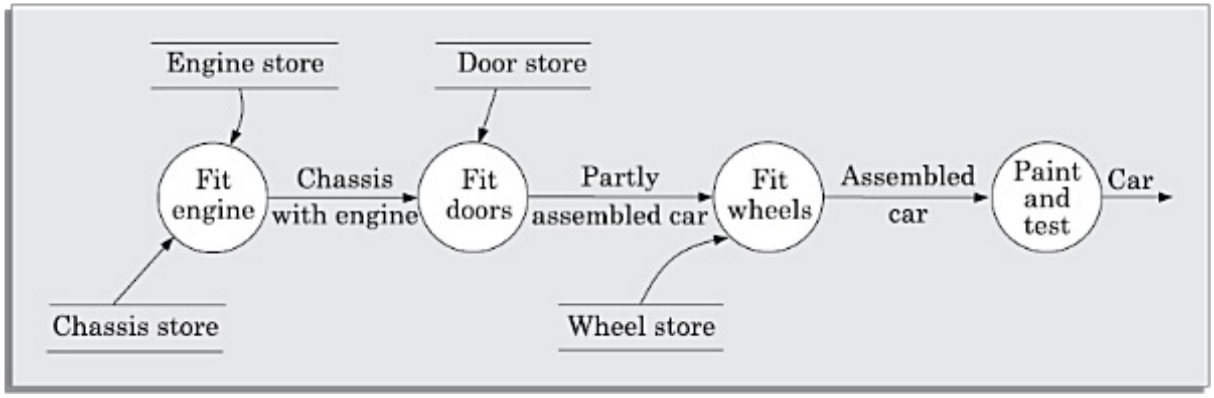

Figure 1.11: Data flow model of a car assembly plant.

1.4.6 Object-oriented Design

Data flow-oriented techniques evolved into object-oriented design (OOD) techniques in the late seventies. Object-oriented design technique is an intuitively appealing approach, where the natural objects (such as employees, pay-roll-register, etc.) relevant to a problem are first identified and then the relationships among the objects such as composition, reference, and inheritance are determined. Each object essentially acts as a *data hiding* (also known as *data abstraction*) entity. Object-oriented techniques have gained wide spread acceptance because of their simplicity, the scope for code and design reuse, promise of lower development time, lower development cost, more robust code, and easier maintenance. OOD techniques are discussed in Chapters 7 and 8.

1.4.7 What Next?

In this section, we have so far discussed how software design techniques have evolved since the early days of programming. We pictorially summarise this evolution of the software design techniques in Figure 1.12. It can be observed that in almost every passing decade, revolutionary ideas were put forward to design larger and more sophisticated programs, and at the same time the quality of the design solutions improved. But, what would the next improvement to the design techniques be? It is very difficult to speculate about the developments that may occur in the future. However, we have already seen that in the past, the design techniques have evolved each time to meet the challenges faced in developing contemporary software. Therefore, the next development would most probably occur to help meet the challenges being faced by the modern software designers. To get an indication of the techniques that are likely to emerge, let us first examine what are the current challenges in designing software. First, program sizes are further increasing as compared to what was being developed a decade back. Second, many of the present day software are required to work in a client-server environment through a web browser-based access (called *web-based software*). At the same time, embedded devices are experiencing an unprecedented growth and rapid customer acceptance in the last decade. It is there for necessary for developing applications for small hand held devices and embedded processors. We examine later in this text how aspect-oriented programming, client- server based design, and embedded software design techniques have emerged rapidly. In the current decade, service-orientation has emerged as a recent direction of software engineering due to the popularity of web-based applications and public clouds.

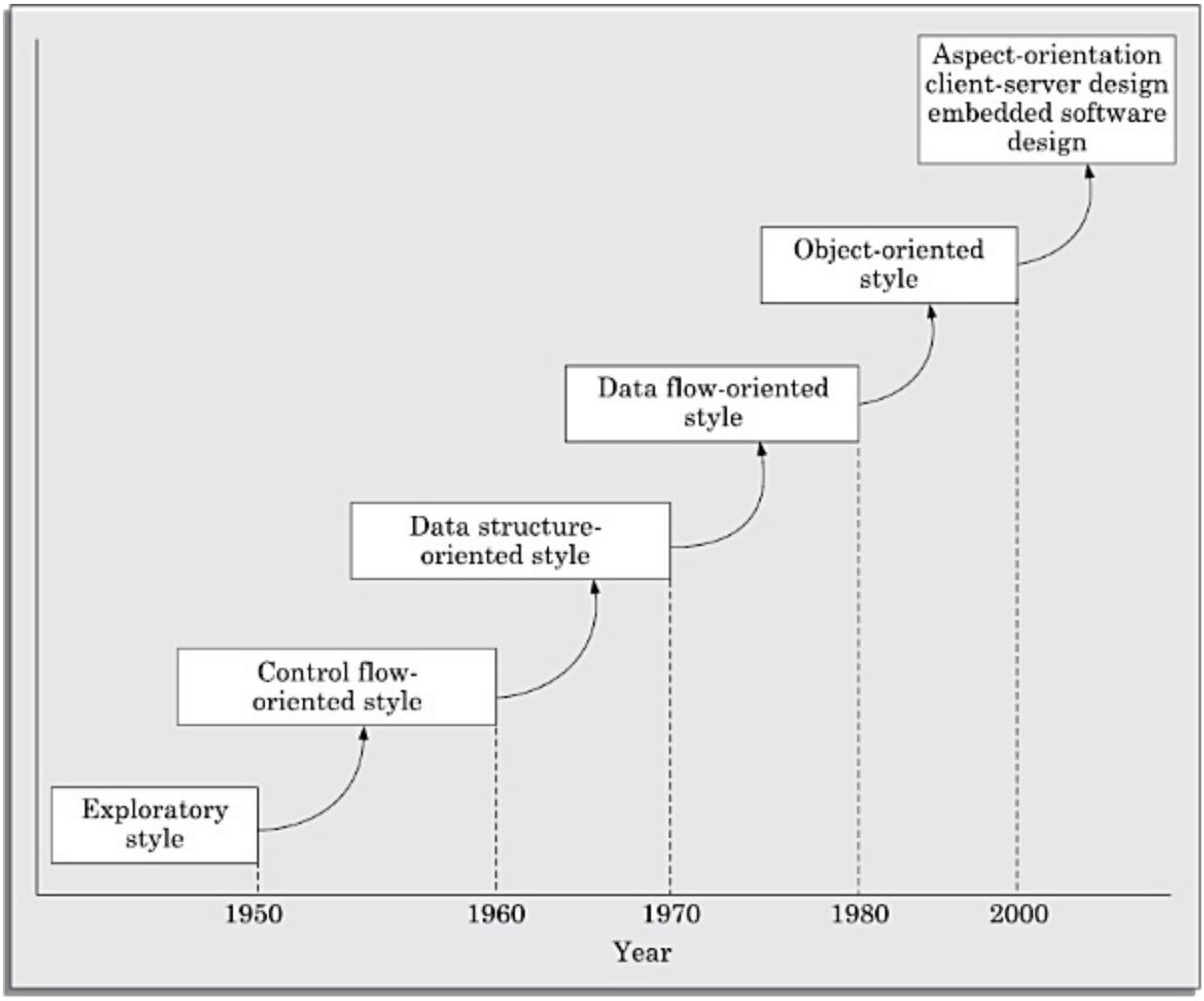

Figure 1.12: Evolution of software design techniques.

1.4.8 Other Developments

It can be seen that remarkable improvements to the prevalent software design technique occurred almost every passing decade. The improvements to the software design methodologies over the last five decades have indeed been remarkable. In addition to the advancements made to the software design techniques, several other new concepts and techniques for effective software development were also introduced. These new techniques include life cycle models, specification techniques, project management techniques, testing techniques, debugging techniques, quality assurance techniques, software measurement techniques, *computer aided software engineering* (CASE) tools, etc. The development of these techniques accelerated the growth of software engineering as a discipline. We shall discuss these techniques in the later chapters.

1.5 NOTABLE CHANGES IN SOFTWARE DEVELOPMENT PRACTICES

Before we discuss the details of various software engineering principles, it is worthwhile to examine the glaring differences that you would notice when you observe an exploratory style of software development and another development effort based on modern software engineering practices. The following noteworthy differences between these two software development approaches would be immediately observable.

- An important difference is that the exploratory software development style is based on *error correction (build and fix)* while the software engineering techniques are based on the principles of *error prevention*. Inherent in the software engineering principles is the realisation that it is much more cost-effective to prevent errors from occurring than to correct them as and when they are detected. Even when mistakes are committed during development, software engineering principles emphasize detection of errors as detected only during the final product testing. In contrast, the modern practice of software development is to develop the software through several well-defined stages such as requirements specification, design, coding, testing, etc., and attempts are made to detect and fix as many errors as possible in the same phase in which they are made.
- In the exploratory style, coding was considered synonymous with software development. For instance, this naive way of developing a software believed in developing a working system as quickly as possible and then successively modifying it until it performed satisfactorily. Exploratory programmers literally dive at the computer to get started with their programs even before they fully learn about the problem!!! It was recognised that exploratory programming not only turns out to be prohibitively costly for non-trivial problems, but also produces hard-to-maintain programs. Even minor modifications to such programs later can become nightmarish. In the modern software development style, coding is regarded as only a small part of the overall software development activities. There are several development activities such as design and testing which may demand much more effort than coding.
- A lot of attention is now being paid to requirements specification. Significant effort is being devoted to develop a clear and correct

specification of the problem before any development activity starts. Unless the requirements specification is able to correctly capture the exact customer requirements, large number of rework would be necessary at a later stage. Such rework would result in higher cost of development and customer dissatisfaction.

- Now there is a distinct design phase where standard design techniques are employed to yield coherent and complete design models.
- Periodic reviews are being carried out during all stages of the development process. The main objective of carrying out reviews is *phase containment of errors*, i.e. detect and correct errors as soon as possible. Phase containment of errors is an important software engineering principle. We will discuss this technique in Chapter 2.
- Today, software testing has become very systematic and standard testing techniques are available. Testing activity has also become all encompassing, as test cases are being developed right from the requirements specification stage.
- There is better visibility of the software through various developmental activities.
-

> By visibility we mean production of good quality, consistent and peer reviewed documents at the end of every software development activity.

- In the past, very little attention was being paid to producing good quality and consistent documents. In the exploratory style, the design and test activities, even if carried out (in whatever way), were not documented satisfactorily. Today, consciously good quality documents are being developed during software development. This has made fault diagnosis and maintenance far more smoother. We will see in Chapter 3 that in addition to facilitating product maintenance, increased visibility makes management of a software project easier.
- Now, projects are being thoroughly planned. The primary objective of project planning is to ensure that the various development activities take place at the correct time and no activity is halted due to the want of some resource. Project planning normally includes preparation of various types of estimates, resource scheduling, and development of project tracking plans. Several techniques and automation tools for tasks such as configuration management, cost estimation, scheduling,

etc., are being used for effective software project management.

- Several metrics (quantitative measurements) of the products and the product development activities are being collected to help in software project management and software quality assurance.

1.6 COMPUTER SYSTEMS ENGINEERING

In all the discussions so far, we assumed that the software being developed would run on some general-purpose hardware platform such as a desktop computer or a server. But, in several situations it may be necessary to develop special hardware on which the software would run. Examples of such systems are numerous, and include a robot, a factory automation system, and a cell phone. In a cell phone, there is a special processor and other specialised devices such as a speaker and a microphone. It can run only the programs written specifically for it. Development of such systems entails development of both software and specific hardware that would run the software. Computer systems engineering addresses development of such systems requiring development of both software and specific hardware to run the software. Thus, systems engineering encompasses software engineering.

The general model of systems engineering is shown schematically in Figure 1.13. One of the important stages in systems engineering is the stage in which decision is made regarding the parts of the problems that are to be implemented in hardware and the ones that would be implemented in software. This has been represented by the box captioned hardware-software partitioning in Figure 1.13. While partitioning the functions between hardware and software, several trade-offs such as flexibility, cost, speed of operation, etc., need to be considered. The functionality implemented in hardware run faster. On the other had, functionalities implemented in software is easier to extend. Further, it is difficult to implement complex functions in hardware. Also, functions implemented in hardware incur extra space, weight, manufacturing cost, and power overhead.

After the hardware-software partitioning stage, development of hardware and software are carried out concurrently (shown as concurrent branches in Figure 1.13). In system engineering, testing the software during development becomes a tricky issue, the hardware on which the software would run and tested would still be under development—remember that the hardware and the software are being developed at the same time. To test the software

during development, it usually becomes necessary to develop simulators that mimic the features of the hardware being developed. The software is tested using these simulators. Once both hardware and software development are complete, these are integrated and tested. The project management activity is required through out the duration of system development as shown in Figure 1.13. In this text, we have confined our attention to software engineering only.

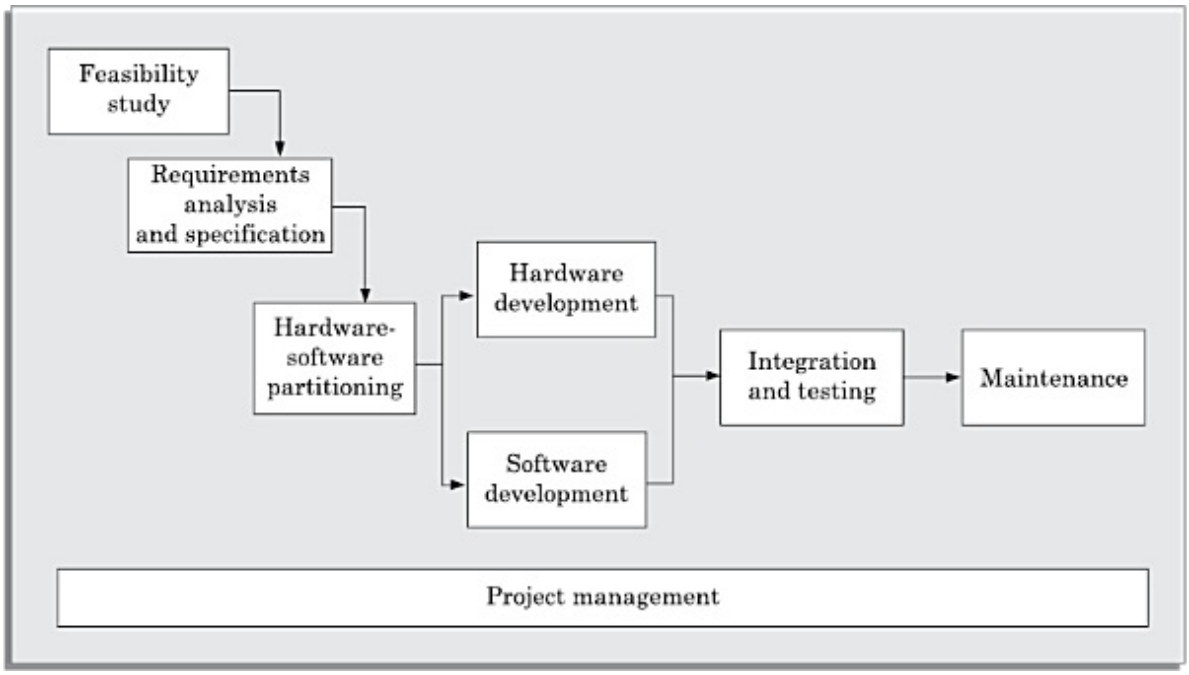

Figure 1.13: Computer systems engineering.

SUMMARY

- We first defined the scope of software engineering. We came up with two alternate but equivalent definitions:

  – The systematic collection of decades of programming experience together with the innovations made by researchers towards developing high quality software in a cost- effective manner.
  – The engineering approach to develop software.

- The exploratory (also called build and fix) style of program development is used by novice programmers. The exploratory style is characterized by quickly developing the program code and then modifying it successively till the program works. This approach turns out not only to be a very costly and inefficient way of developing

software, but yields a product that is unreliable and difficult to maintain. Also, the exploratory style is very difficult to use when software is developed through team effort. A still larger handicap of the exploratory style of programming is that it breaks down when used to develop large programs.

- Unless one makes use of software engineering principles, the increase in effort and time with size of the program would be exponential—making it virtually impossible for a person to develop large programs.
- To handle complexity in a problem, all software engineering principles make extensive use of the following two techniques:

  – Abstraction (modelling), and
  – Decomposition (Divide and conquer).

- Software engineering techniques are essential for development of large software products where a group of engineers work in a team to develop the product. However, most of the principles of software engineering are useful even while developing small programs.
- A program is called structured, when it is decomposed into a set of modules and each module in turn is decomposed into functions. Additionally, structured programs avoid the use of GO TO statements and use only structured programming constructs.
- Computer systems engineering deals with the development of complete systems, necessitating integrated development of both software and hardware parts. Computer systems engineering encompasses software engineering.
- We shall delve into various software engineering principles starting from the next chapter. But, before that here is a word of caution. Those who have written large-sized programs, can better appreciate many of the principles of software engineering. Students with less or no programming experience would have to take our words for it and work harder with the topics. However, it is a fact that unless somebody has seen an elephant (read problems encountered during program development) at least once, any amount of describing and explaining would not result in the kind of understanding that somebody who has previously seen an elephant (developed a program) would get.

EXERCISES

1. Choose the correct option:
   (a) Which of the following is not a symptom of the present software crisis:
      (i) Software is expensive.
      (ii) It takes too long to build a software product.
      (iii) Software is delivered late.
      (iv) Software products are required to perform very complex tasks.
   (b) The goal of structured programming is which one of the following:
      (i) To have well indented programs.
      (ii) To be able to infer the flow of control from the compiled code.
      (iii) To be able to infer the flow of control from the program text.
      (iv) To avoid the use of GO TO statements.
   (c) Unrestricted use of GO TO statements is normally avoided while writing a program, since:
      (i) It increases the running time of programs.
      (ii) It increases memory requirements of programs.
      (iii) It results in larger executable code sizes.
      (iv) It makes debugging difficult.
   (d) Why is writing easily modifiable code important?
      (i) Easily modifiable code results in quicker run time.
      (ii) Most real world programs require change at some point of time or other.
      (iii) Most text editors make it mandatory to write modifiable code.
      (iv) Several people may be writing different parts of code at the same time.
2. What is the principal aim of the software engineering discipline? What does the discipline of software engineering discuss?
3. Why do you think systematic software development using the software engineering principle is any different than art or craft?
4. Distinguish between a program and a professionally developed software.
5. Distinguish among a program, a software product and a software service. Give one example of each. Discuss the difference of the characteristics of development projects for each of these.
6. What is a software product line? Give an example of a software product line. How is a software product line development any different from a software product development.
7. What are the main types of projects that are undertaken by software

development companies? Give examples of these types of projects and point out the important characteristic differences between these types of projects.

8. Do you agree with the following statement—The focus of exploratory programming is error correction while the software engineering principles emphasise error prevention"? Give the reasonings behind your answer.
9. What difficulties would a software development company face, if it tries to use the exploratory (build and fix) program development style in its development projects? Explain your answer.
10. What are the symptoms of the present software crisis? What factors have contributed to the making of the present software crisis? What are the possible solutions to the present software crisis?
11. Explain why the effort, time, and cost required to develop a program using the build and fix style increase exponentially with the size of the program? How do software engineering principles help tackle this rapid rise in development time and cost?
12. Distinguish between software products and services. Give examples of each.
13. What are the different types of projects that are being undertaken by software development houses? Which of these type of projects is the forte of Indian software development organisations? Identify any possible reasons as to why the other has not been focused by the Indian software development organisations.
14. Name the basic techniques used by the software engineering techniques to handle complexity in a problem.
15. What do you understand by the exploratory (also known as the build and fix) style of software development? Graphically depict the activities that a programmer typically carries out while developing a programming solution using the exploratory style. In your diagram also show the order in which the activities are carried out. What are the shortcomings of this style of program development?
16. List the major differences between the exploratory and modern software development practices.
17. What is the difference between the actual complexity of solving a problem and its perceived complexity? What causes the difference between the two to arise?
18. What do you understand by the term perceived complexity of a problem? How is it different from computational complexity? How can the

perceived complexity of a problem be reduced?

19. Why is the number 7 considered as a magic number in software engineering? How is it useful software engineering?
20. What do you understand by the principles of abstraction and decomposition? Why are these two principles considered important in software engineering? Explain the problems that these two principles target to solve? Support your answer using suitable examples.
21. What do you understand by control flow structure of a program? Why is it difficult to understand a program having a messy control flow structure? How can a good control flow structure for a program be designed?
22. What is a flow chart? How is the flow charting technique useful during software development?
23. What do you understand by visibility of design and code? How does increased visibility help in systematic software development? (We shall revisit this question in Chapter 3)
24. What do you understand by the term—structured programming? How do modern programming languages such as PASCAL and C facilitate writing structured programs? What are the advantages of writing structured programs vis-a-vis unstructured programs?
25. What is a high-level programming language? Why does a programmer using a high-level programming language have a higher productivity as compared to when using machine language for application development?
26. What are the three basic types of program constructs necessary to develop the program for any given problem? Give examples of these three constructs from any high-level language you know.
27. What do you understand by a program module? What are the important characteristics of a program module?
28. Explain how do the use of software engineering principles help to develop software products cost-effectively and timely. Elaborate your answer by using suitable examples.
29. What is the basic difference between a control flow-oriented and a data flow-oriented design technique? Can you think of any reason as to why a data flow-oriented design technique is likely to produce better designs than a control flow-oriented design technique? (We shall revisit this question while discussing the design techniques in Chapter 6.)
30. Name the two fundamental principles that are used extensively in software engineering to tackle the complexity in developing large programs? Explain these two principles. By using suitable examples

explain how these two principles help tackle the complexity associated with developing large programs.

31. What does the *control flow graph* (CFG) of a program represent? Draw the CFG of the following program:

```
main(){
  int y=1;
  if(y<0)
    if(y>0) y=3;
    else y=0;
  printf("%d\n",y);
}
```

32. Discuss the possible reasons behind supersession of the data structure-oriented design methods by the control flow-oriented design methods.
33. What is a data structure-oriented software design methodology? How is it different from the data flow-oriented design methodology?
34. Discuss the major advantages of the *object-oriented design* (OOD) methodologies over the data flow-oriented design methodologies.
35. Explain how the software design techniques have evolved in the past. How do you think shall the software design techniques evolve in the near future?
36. What is computer systems engineering? How is it different from software engineering?
    Give examples of some types of product development projects for which systems engineering is appropriate.
37. What do you mean by software service? Explain the important differences between the characteristics of a software service development project and a software product development project.

1 In this text, we shall use the terms *module* and *module structure* to loosely mean the following—A *module* is a collection of procedures and data structures. The data structures in a module are accessible only to the procedures defined inside the module. A module forms an independently compilable unit and may be linked to other modules to form a complete application. The term *module structure* will be used to denote the way in which different modules invoke each other's procedures.

Chapter 2

SOFTWARE LIFE CYCLE MODELS

In Chapter 1, we discussed a few basic issues in software engineering. We pointed out a few important differences between the exploratory program development style and the software engineering approach. Please recollect from our discussions in Chapter 1 that the exploratory style is also known as the *build* and *fix* programming. In build and fix programming, a programmer typically starts to write the program immediately after he has formed an informal understanding of the requirements. Once program writing is complete, he gets down to fix anything that does not meet the user's expectations. Usually, a large number of code fixes are required even for toy programs. This pushes up the development costs and pulls down the quality of the program. Further, this approach usually turns out to be a recipe for project failure when used to develop non-trivial programs requiring team effort. In contrast to the build and fix style, the software engineering approaches emphasise software development through a well-defined and ordered set of activities. These activities are graphically modelled (represented) as well as textually described and are variously called as *software life cycle model*, *software development life cycle* (SDLC) *model*, and *software development process model*. Several life cycle models have so far been proposed. However, in this Chapter we confine our attention to only a few important and commonly used ones.

In this chapter, we first discuss a few basic concepts associated with life cycle models. Subsequently, we discuss the important activities that have been prescribed to be carried out in the classical waterfall model. This is intended to provide an insight into the activities that are carried out as part of every life cycle model. In fact, the classical waterfall model can be considered as a basic model and all other life cycle models as extensions of this model to cater to specific project situations. After discussing the waterfall

model, we discuss a few derivatives of this model. Subsequently we discuss the spiral model that generalises various life cycle models. Finally, we discuss a few recently proposed life cycle models that are categorized under the umbrella term *agile model*. Of late, agile models are finding increasing acceptance among developers and researchers.

The genesis of the agile model can be traced to the radical changes to the types of project that are being undertaken at present, rather than to any radical innovations to the life cycle models themselves. The projects have changed from large multi-year product development projects to small services projects now

.

2.1 A FEW BASIC CONCEPTS

In this section, we present a few basic concepts concerning the life cycle models.

Software life cycle

It is well known that all living organisms undergo a life cycle. For example when a seed is planted, it germinates, grows into a full tree, and finally dies. Based on this concept of a biological life cycle, the term *software life cycle* has been defined to imply the different stages (or phases) over which a software evolves from an initial customer request for it, to a fully developed software, and finally to a stage where it is no longer useful to any user, and then it is discarded.

As we have already pointed out, the life cycle of every software starts with a request for it by one or more customers. At this stage, the customers are usually not clear about all the features that would be needed, neither can they completely describe the identified features in concrete terms, and can only vaguely describe what is needed. This stage where the customer feels a need for the software and forms rough ideas about the required features is known as the *inception* stage. Starting with the inception stage, a software evolves through a series of identifiable stages (also called phases) on account of the development activities carried out by the developers, until it is fully developed and is released to the customers.

Once installed and made available for use, the users start to use the software. This signals the start of the operation (also called *maintenance*) phase. As the users use the software, not only do they request for fixing any failures that they might encounter, but they also continually suggest several

improvements and modifications to the software. Thus, the maintenance phase usually involves continually making changes to the software to accommodate the bug-fix and change requests from the user. The operation phase is usually the longest of all phases and constitutes the useful life of a software. Finally the software is retired, when the users do not find it any longer useful due to reasons such as changed business scenario, availability of a new software having improved features and working, changed computing platforms, etc. This forms the essence of the life cycle of every software. Based on this description, we can define the software life cycle as follows:

> The life cycle of a software represents the series of identifiable stages through which it evolves during its life time.

With this knowledge of a software life cycle, we discuss the concept of a software life cycle model and explore why it is necessary to follow a life cycle model in professional software development environments.

Software development life cycle (SDLC) model

In any systematic software development scenario, certain well-defined activities need to be performed by the development team and possibly by the customers as well, for the software to evolve from one stage in its life cycle to the next. For example, for a software to evolve from the requirements specification stage to the design stage, the developers need to elicit requirements from the customers, analyse those requirements, and formally document the requirements in the form of an SRS document.

A *software development life cycle* (SDLC) model (also called *software life cycle model* and *software development process model*) describes the different activities that need to be carried out for the software to evolve in its life cycle. Throughout our discussion, we shall use the terms *software development life cycle* (SDLC) and *software development process* interchangeably. However, some authors distinguish an SDLC from a software development process. In their usage, a software development process describes the life cycle activities more precisely and elaborately, as compared to an SDLC. Also, a development process may not only describe various activities that are carried out over the life cycle, but also prescribe a specific methodologies to carry out the activities, and also recommends the the specific documents and other artifacts that should be produced at the end of each phase. In this sense, the term SDLC can be considered to be a more generic term, as compared to the

development process and several development processes may fit the same SDLC.

An SDLC is represented graphically by drawing various stages of the life cycle and showing the transitions among the phases. This graphical model is usually accompanied by a textual description of various activities that need to be carried out during a phase before that phase can be considered to be complete. In simple words, we can define an SDLC as follows:

> An SDLC graphically depicts the different phases through which a software evolves. It is usually accompanied by a textual description of the different activities that need to be carried out during each phase.

Process *versus* methodology

Though the terms *process* and *methodology* are at time used interchangeably, there is a subtle difference between the two. First, the term process has a broader scope and addresses either all the activities taking place during software development, or certain coarse grained activities such as design (e.g. design process), testing (test process), etc. Further, a software process not only identifies the specific activities that need to be carried out, but may also prescribe certain methodology for carrying out each activity. For example, a design process may recommend that in the design stage, the high-level design activity be carried out using Hatley and Pirbhai's structured analysis and design methodology. A methodology, on the other hand, prescribes a set of steps for carrying out a specific life cycle activity. It may also include the rationale and philosophical assumptions behind the set of steps through which the activity is accomplished.

> A software development process has a much broader scope as compared to a software development methodology. A process usually describes all the activities starting from the inception of a software to its maintenance and retirement stages, or at least a chunk of activities in the life cycle. It also recommends specific methodologies for carrying out each activity. A methodology, in contrast, describes the steps to carry out only a single or at best a few individual activities.

Why use a development process?

The primary advantage of using a development process is that it encourages development of software in a systematic and disciplined manner. Adhering to a process is especially important to the development of professional software needing team effort. When software is developed by a team rather than by

an individual programmer, use of a life cycle model becomes indispensable for successful completion of the project.

Software development organisations have realised that adherence to a suitable life cycle model helps to produce good quality software and that helps minimise the chances of time and cost overruns.

Suppose a single programmer is developing a small program. For example, a student may be developing code for a class room assignment. The student might succeed even when he does not strictly follow a specific development process and adopts a build and fix style of development. However, it is a different ball game when a professional software is being developed by a team of programmers. Let us now understand the difficulties that may arise if a team does not use any development process, and the team members are given complete freedom to develop their assigned part of the software as per their own discretion. Several types of problems may arise. We illustrate one of the problems using an example. Suppose, a software development problem has been divided into several parts and these parts are assigned to the team members. From then on, suppose the team members are allowed the freedom to develop the parts assigned to them in whatever way they like. It is possible that one member might start writing the code for his part while making assumptions about the input results required from the other parts, another might decide to prepare the test documents first, and some other developer might start to carry out the design for the part assigned to him. In this case, severe problems can arise in interfacing the different parts and in managing the overall development. Therefore, *ad hoc* development turns out to be is a sure way to have a failed project. Believe it or not, this is exactly what has caused many project failures in the past!

When a software is developed by a team, it is necessary to have a precise understanding among the team members as to—when to do what. In the absence of such an understanding, if each member at any time would do whatever activity he feels like doing. This would be an open invitation to developmental chaos and project failure. The use of a suitable life cycle model is crucial to the successful completion of a team-based development project. But, do we need an SDLC model for developing a small program. In this context, we need to distinguish between programming-in-the-small and programming-in-the-large.

Programming-in-the-small refers to development of a toy program by a single programmer. Whereas programming-in-the-large refers to development of a

professional software through team effort. While development of a software of the former type could succeed even while an individual programmer uses a build and fix style of development, use of a suitable SDLC is essential for a professional software development project involving team effort to succeed.

Why document a development process?

It is not enough for an organisation to just have a well-defined development process, but the development process needs to be properly documented. To understand the reason for this, let us consider that a development organisation does not document its development process. In this case, its developers develop only an informal understanding of the development process. An informal understanding of the development process among the team members can create several problems during development. We have identified a few important problems that may crop up when a development process is not adequately documented. Those problems are as follows:

- A documented process model ensures that every activity in the life cycle is accurately defined. Also, wherever necessary the methodologies for carrying out the respective activities are described. Without documentation, the activities and their ordering tend to be loosely defined, leading to confusion and misinterpretation by different teams in the organisation. For example, code reviews may informally and inadequately be carried out since there is no documented methodology as to how the code review should be done. Another difficulty is that for loosely defined activities, the developers tend to use their subjective judgments. As an example, unless it is explicitly prescribed, the team members would subjectively decide as to whether the test cases should be designed just after the requirements phase, after the design phase, or after the coding phase. Also, they would debate whether the test cases should be documented at all and the rigour with it should be documented.
- An undocumented process gives a clear indication to the members of the development teams about the lack of seriousness on the part of the management of the organisation about following the process. Therefore, an undocumented process serves as a hint to the developers to loosely follow the process. The symptoms of an undocumented process are easily visible—designs are shabbily done, reviews are not carried out rigorously, etc.

- A project team might often have to tailor a standard process model for use in a specific project. It is easier to tailor a documented process model, when it is required to modify certain activities or phases of the life cycle. For example, consider a project situation that requires the testing activities to be outsourced to another organisation. In this case, A documented process model would help to identify where exactly the required tailoring should occur.
- A documented process model, as we discuss later, is a mandatory requirement of the modern quality assurance standards such as ISO 9000 and SEI CMM. This means that unless a software organisation has a documented process, it would not qualify for accreditation with any of the quality standards. In the absence of a quality certification for the organisation, the customers would be suspicious of its capability of developing quality software and the organisation might find it difficult to win tenders for software development.

A documented development process forms a common understanding of the activities to be carried out among the software developers and helps them to develop software in a systematic and disciplined manner. A documented development process model, besides preventing the misinterpretations that might occur when the development process is not adequately documented, also helps to identify inconsistencies, redundancies, and omissions in the development process.

Nowadays, good software development organisations normally document their development process in the form of a booklet. They expect the developers recruited fresh to their organisation to first master their software development process during a short induction training that they are made to undergo.

Phase entry and exit criteria

A good SDLC besides clearly identifying the different phases in the life cycle, should unambiguously define the entry and exit criteria for each phase. The phase entry (or exit) criteria is usually expressed as a set of conditions that needs to be be satisfied for the phase to start (or to complete). As an example, the phase exit criteria for the software requirements specification phase, can be that the *software requirements specification* (SRS) document is ready, has been reviewed internally, and also has been reviewed and approved by the customer. Only after these criteria are satisfied, the next phase can start.

If the entry and exit criteria for various phases are not well-defined, then that would leave enough scope for ambiguity in starting and ending various phases, and cause lot of confusion among the developers. Sometimes they might prematurely stop the activities in a phase, and some other times they might continue working on a phase much after when the phase should have been over. The decision regarding whether a phase is complete or not becomes subjective and i t becomes difficult for the project manager to accurately tell how much has the development progressed. When the phase entry and exit criteria are not well-defined, the developers might close the activities of a phase much before they are actually complete, giving a false impression of rapid progress. In this case, it becomes very difficult for the project manager to determine the exact status of development and track the progress of the project. This usually leads to a problem that is usually identified as the *99 per cent complete syndrome*. This syndrome appears when there the software project manager has no definite way of assessing the progress of a project, the optimistic team members feel that their work is 99 per cent complete even when their work is far from completion—making all projections made by the project manager about the project completion time to be highly inaccurate.

2.2 WATERFALL MODEL AND ITS EXTENSIONS

The waterfall model and its derivatives were extremely popular in the 1970s and still are heavily being used across many development projects. The waterfall model is possibly the most obvious and intuitive way in which software can be developed through team effort. We can think of the waterfall model as a generic model that has been extended in many ways for catering to certain specific software development situations to realise all other software life cycle models. For this reason, after discussing the classical and iterative waterfall models, we discuss its various extensions.

2.2.1 Classical Waterfall Model

Classical waterfall model is intuitively the most obvious way to develop software. It is simple but idealistic. In fact, it is hard to put this model into use in any non-trivial software development project. One might wonder if this model is hard to use in practical development projects, then why study it at all? The reason is that all other life cycle models can be thought of as being extensions of the classical waterfall model.

Therefore, it makes sense to first understand the classical waterfall model, in order to be able to develop a proper understanding of other life cycle models. Besides, we shall see later in this text that this model though not used for software development; is implicitly used while documenting software.

The classical waterfall model divides the life cycle into a set of phases as shown in Figure 2.1. It can be easily observed from this figure that the diagrammatic representation of the classical waterfall model resembles a multi-level waterfall. This resemblance justifies the name of the model.

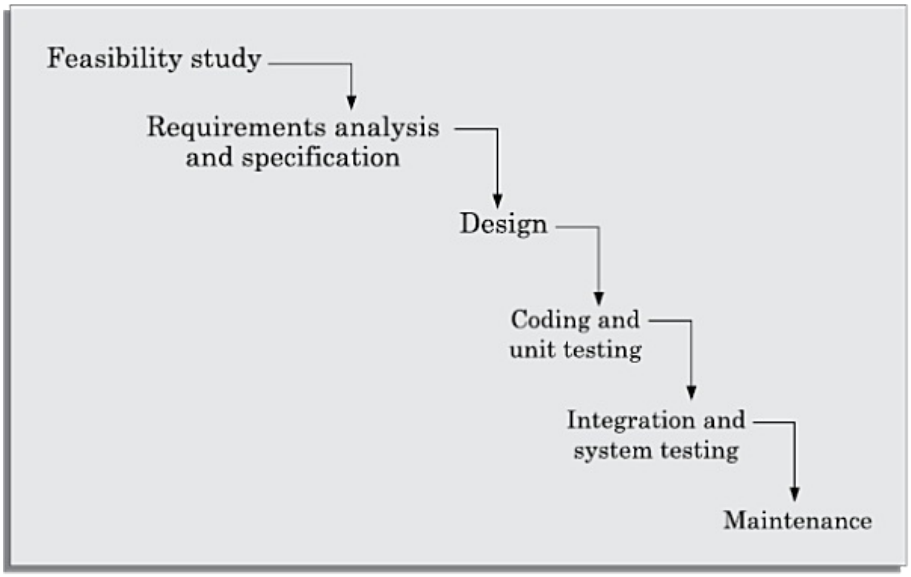

Figure 2.1: Classical waterfall model.

Phases of the classical waterfall model

The different phases of the classical waterfall model have been shown in Figure 2.1. As shown in Figure 2.1, the different phases are—feasibility study, requirements analysis and specification, design, coding and unit testing, integration and system testing, and maintenance. The phases starting from the feasibility study to the integration and system testing phase are known as the *development phases.* A software is developed during the development phases, and at the completion of the development phases, the software is delivered to the customer. After the delivery of software, customers start to use the software signalling the commencement of the operation phase. As the customers start to use the software, changes to it become necessary on account of bug fixes and feature extensions, causing maintenance works to be undertaken. Therefore, the last phase is also known as the *maintenance phase* of the life cycle. It needs to be kept in mind that some of the text

books have different number and names of the phases.

An activity that spans all phases of software development is *project management*. Since it spans the entire project duration, no specific phase is named after it. Project management, nevertheless, is an important activity in the life cycle and deals with managing the software development and maintenance activities.

In the waterfall model, different life cycle phases typically require relatively different amounts of efforts to be put in by the development team. The relative amounts of effort spent on different phases for a typical software has been shown in Figure 2.2. Observe from Figure 2.2 that among all the life cycle phases, the maintenance phase normally requires the maximum effort. On the average, about 60 per cent of the total effort put in by the development team in the entire life cycle is spent on the maintenance activities alone.

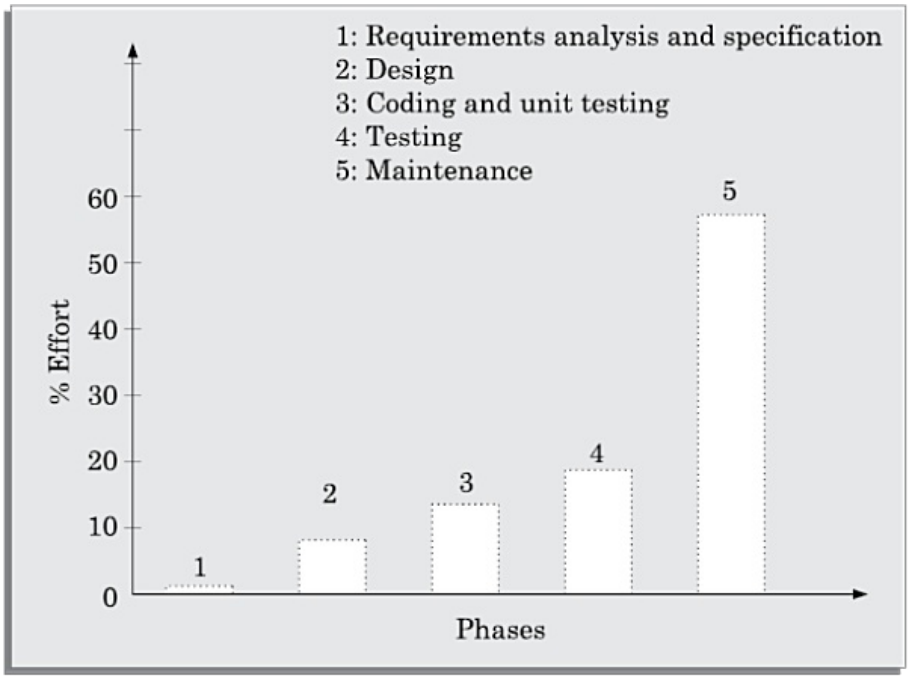

Figure 2.2: Relative effort distribution among different phases of a typical product.

However, among the development phases, the integration and system testing phase requires the maximum effort in a typical development project. In the following subsection, we briefly describe the activities that are carried out in the different phases of the classical waterfall model.

Feasibility study

The main focus of the feasibility study stage is to determine whether it would be *financially* and *technically feasible* to develop the software. The

feasibility study involves carrying out several activities such as collection of basic information relating to the software such as the different data items that would be input to the system, the processing required to be carried out on these data, the output data required to be produced by the system, as well as various constraints on the development. These collected data are analysed to perform at the following:

Development of an overall understanding of the problem: It is necessary to first develop an overall understanding of what the customer requires to be developed. For this, only the the important requirements of the customer need to be understood and the details of various requirements such as the screen layouts required in the *graphical user interface* (GUI), specific formulas or algorithms required for producing the required results, and the databases schema to be used are ignored.

Formulation of the various possible strategies for solving the problem: In this activity, various possible high-level solution schemes to the problem are determined. For example, solution in a client-server framework and a standalone application framework may be explored.

Evaluation of the different solution strategies: The different identified solution schemes are analysed to evaluate their benefits and shortcomings. Such evaluation often requires making approximate estimates of the resources required, cost of development, and development time required. The different solutions are compared based on the estimations that have been worked out. Once the best solution is identified, all activities in the later phases are carried out as per this solution. At this stage, it may also be determined that none of the solutions is feasible due to high cost, resource constraints, or some technical reasons. This scenario would, of course, require the project to be abandoned.

We can summarise the outcome of the feasibility study phase by noting that other than deciding whether to take up a project or not, at this stage very high-level decisions regarding the solution strategy is defined. Therefore, feasibility study is a very crucial stage in software development. The following is a case study of the feasibility study undertaken by an organisation. It is intended to give a feel of the activities and issues involved in the feasibility study phase of a typical software project.

Case study 2.1

A mining company named Galaxy Mining Company Ltd. (GMC Ltd.) has mines located at various places in India. It has about fifty different mine sites spread across eight states. The company employs a large number of miners at each mine site. Mining being a risky profession, the company intends to operate a special provident fund, which would exist in addition to the standard provident fund that the miners already enjoy. The main objective of having the special provident fund (SPF) would be to quickly distribute some compensation before the PF amount is paid.

According to this scheme, each mine site would deduct SPF installments from each miner every month and deposit the same to the central special provident fund commissioner (CSPFC). The CSPFC will maintain all details regarding the SPF installments collected from the miners.

GMC Ltd. requested a reputed software vendor Adventure Software Inc. to undertake the task of developing the software for automating the maintenance of SPF records of all employees. GMC Ltd. has realised that besides saving manpower on book-keeping work, the software would help in speedy settlement of claim cases. GMC Ltd. indicated that the amount it can at best afford Rs. 1 million for this software to be developed and installed.

Adventure Software Inc. deputed their project manager to carry out the feasibility study. The project manager discussed with the top managers of GMC Ltd. to get an overview of the project. He also discussed with the field PF officers at various mine sites to determine the exact details of the project. The project manager identified two broad approaches to solve the problem. One is to have a central database which would be accessed and updated via a satellite connection to various mine sites. The other approach is to have local databases at each mine site and to update the central database periodically through a dial-up connection. This periodic updates can be done on a daily or hourly basis depending on the delay acceptable to GMC Ltd. in invoking various functions of the software. H e found that the second approach is very affordable and more fault-tolerant as the local mine sites can operate even when the communication link temporarily fails. In this approach, when a link fails, only the update of the central database gets delayed. Whereas in the first approach, all SPF work gets stalled at a mine site for the entire duration of link failure. The project manager quickly analysed t h e overall database functionalities required, the user-interface issues, and the software handling communication with the mine sites. From this analysis, he estimated the approximate cost to develop the software. He found that a solution involving maintaining local databases at the mine sites and periodically updating a central database is financially and technically feasible. The project manager discussed this solution with the president of GMC Ltd., who indicated that the proposed solution would be acceptable to them.

Requirements analysis and specification

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document

them properly. This phase consists of two distinct activities, namely requirements gathering and analysis, and requirements specification. In the following subsections, we give an overview of these two activities:

- **Requirements gathering and analysis:** The goal of the requirements gathering activity is to collect all relevant information regarding the software to be developed from the customer with a view to clearly understand the requirements. For this, first requirements are gathered from the customer and then the gathered requirements are analysed. The goal of the requirements analysis activity is to weed out the incompleteness and inconsistencies in these gathered requirements. Note that a n *inconsistent* requirement is one in which some part of the requirement contradicts with some other part. On the other hand, a n *incomplete* requirement is one in which some parts of the actual requirements have been omitted.
- **Requirements specification:** After the requirement gathering and analysis activities are complete, the identified requirements are documented. This is called a *software requirements specification* (SRS) document. The SRS document is written using end-user terminology. This makes the SRS document understandable to the customer. Therefore, understandability of the SRS document is an important issue. The SRS document normally serves as a contract between the development team and the customer. Any future dispute between the customer and the developers can be settled by examining the SRS document. The SRS document is therefore an important document which must be thoroughly understood by the development team, and reviewed jointly with the customer. The SRS document not only forms the basis for carrying out all the development activities, but several documents such as users' manuals, system test plan, etc. are prepared directly based on it. In Chapter 4, we examine the requirements analysis activity and various issues involved in developing a good SRS document in more detail.

Design

The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the *software architecture* is derived from the SRS document. Two

distinctly different design approaches are popularly being used at present—the procedural and object-oriented design approaches. In the following, we briefly discuss the essence of these two approaches. These two approaches are discussed in detail in Chapters 6, 7, and 8.

- **Procedural design approach:** The traditional design approach is in use in many software development projects at the present time. This traditional design technique is based on the data flow-oriented design approach. It consists of two important activities; first *structured analysis* of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a *structured design* step where the results of structured analysis are transformed into the software design.

During structured analysis, the functional requirements specified in the SRS document are decomposed into subfunctions and the data-flow among these subfunctions is analysed and represented diagrammatically in the form of DFDs. The DFD technique is discussed in Chapter 6. Structured design is undertaken once the structured analysis activity is complete. Structured design consists of two main activities—architectural design (also called *high-level design*) and detailed design (also called *Low-level design*). High-level design involves decomposing the system into modules, and representing the interfaces and the invocation relationships among the modules. A high-level software design is some times referred to as the *software architecture*. During the detailed design activity, internals of the individual modules such as the data structures and algorithms of the modules are designed and documented.

- **Object-oriented design approach:** In this technique, various objects that occur in the problem domain and the solution domain are first identified and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design. The OOD approach is credited to have several benefits such as lower development time and effort, and better maintainability of the software. The object-oriented design technique is discussed in Chapters 7 and 8.

Coding and unit testing

The purpose of the coding and unit testing phase is to translate a software design into source code and to ensure that individually each function is working correctly. The coding phase is also sometimes called the *implementation phase*, since the design is implemented into a workable solution in this phase. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually unit tested. The main objective of unit testing is to determine the correct working of the individual modules. The specific activities carried out during unit testing include designing test cases, testing, debugging to fix problems, and management of test cases. We shall discuss the coding and unit testing techniques in Chapter 10.

Integration and system testing

Integration of different modules is undertaken soon after they have been coded and unit tested. During the integration and system testing phase, the different modules are integrated in a planned manner. Various modules making up a software are almost never integrated in one shot (can you guess the reason for this?). Integration of various modules are normally carried out incrementally over a number of steps. During each integration step, previously planned modules are added to the partially integrated system and the resultant system is tested. Finally, after all the modules have been successfully integrated and tested, the full working system is obtained. System testing is carried out on this fully working system.

Integration testing is carried out to verify that the interfaces among different units are working satisfactorily. On the other hand, the goal of system testing is to ensure that the developed system conforms to the requirements that have been laid out in the SRS document.

System testing usually consists of three different kinds of testing activities:

- **-testing:** testing is the system testing performed by the development team.
- **-testing:** This is the system testing performed by a friendly set of customers.
- **Acceptance testing:** After the software has been delivered, the customer performs system testing to determine whether to accept the delivered software or to reject it.

We discuss various integration and system testing techniques in more detail in Chapter 10.

Maintenance

The total effort spent on maintenance of a typical software during its operation phase is much more than that required for developing the software itself. Many studies carried out in the past confirm this and indicate that the ratio of relative effort of developing a typical software product and the total effort spent on its maintenance is roughly 40:60. Maintenance is required in the following three types of situations:

- **Corrective maintenance:** This type of maintenance is carried out to correct errors that were not discovered during the product development phase.
- **Perfective maintenance:** This type of maintenance is carried out to improve the performance of the system, or to enhance the functionalities of the system based on customer's requests.
- **Adaptive maintenance:** Adaptive maintenance is usually required for porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system.

Various maintenance activities have been discussed in more detail in Chapter 13.

Shortcomings of the classical waterfall model

The classical waterfall model is a very simple and intuitive model. However, it suffers from several shortcomings. Let us identify some of the important shortcomings of the classical waterfall model:

No feedback paths: In classical waterfall model, the evolution of a software from one phase to the next is analogous to a waterfall. Just as water in a waterfall after having flowed down cannot flow back, once a phase is complete, the activities carried out in it and any artifacts produced in this phase are considered to be final and are closed for any rework. This requires that all activities during a phase are flawlessly carried out.

The classical waterfall model is idealistic in the sense that it assumes that no error is ever committed by the developers during any of the life cycle phases, and therefore, incorporates no mechanism for error correction.

Contrary to a fundamental assumption made by the classical waterfall model, in practical development environments, the developers do commit a large number of errors in almost every activity they carry out during various phases of the life cycle. After all, programmers are humans and as the old adage says *to err is humane*. The cause for errors can be many—oversight, wrong interpretations, use of incorrect solution scheme, communication gap, etc. These defects usually get detected much later in the life cycle. For example, a design defect might go unnoticed till the coding or testing phase. Once a defect is detected at a later time, the developers need to redo some of the work done during that phase and also redo the work of later phases that are affected by the rework. Therefore, in any non-trivial software development project, it becomes nearly impossible to strictly follow the classical waterfall model of software development.

Difficult to accommodate change requests: This model assumes that all customer requirements can be completely and correctly defined at the beginning of the project. There is much emphasis on creating an unambiguous and complete set of requirements. But, it is hard to achieve this even in ideal project scenarios. The customers' requirements usually keep on changing with time. But, in this model it becomes difficult to accommodate any requirement change requests made by the customer after the requirements specification phase is complete, and this often becomes a source of customer discontent.

Inefficient error corrections: This model defers integration of code and testing tasks until it is very late when the problems are harder to resolve.

No overlapping of phases: This model recommends that the phases be carried out sequentially—new phase can start only after the previous one completes. However, it is rarely possible to adhere to this recommendation and it leads to a large number of team members to idle for extended periods. For example, for efficient utilisation of manpower, the testing team might need to design the system test cases immediately after requirements specification is complete. (We shall discuss in Chapter 10 that the system test cases are designed solely based on the SRS document). In this case, the activities of the design and testing phases overlap. Consequently, it is safe to say that in a practical software development scenario, rather than having a precise point in time at which a phase transition occurs, the different phases need to overlap for cost and efficiency reasons.

Is the classical waterfall model useful at all?

We have already pointed out that it is hard to use the classical waterfall model in real projects. In any practical development environment, as the software takes shape, several iterations through the different waterfall stages become necessary for correction of errors committed during various phases. Therefore, the classical waterfall model is hardly usable for software development. But, as suggested by Parnas [1972] the final documents for the product should be written as if the product was developed using a pure classical waterfall.

> Irrespective of the life cycle model that is actually followed for a product development, the final documents are always written to reflect a classical waterfall model of development, so that comprehension of the documents becomes easier for any one reading the document.

The rationale behind preparation of documents based on the classical waterfall model can be explained using Hoare's metaphor of mathematical theorem [1994] proving—A mathematician presents a proof as a single chain of deductions, even though the proof might have come from a convoluted set of partial attempts, blind alleys and backtracks. Imagine how difficult it would be to understand, if a mathematician presents a proof by retaining all the backtracking, mistake corrections, and solution refinements he made while working out the proof.

2.2.2 Iterative Waterfall Model

We had pointed out in the previous section that in a practical software development project, the classical waterfall model is hard to use. We had branded the classical waterfall model as an idealistic model. In this context, the iterative waterfall model can be thought of as incorporating the necessary changes to the classical waterfall model to make it usable in practical software development projects.

> The main change brought about by the iterative waterfall model to the classical waterfall model is in the form of providing feedback paths from every phase to its preceding phases.

The feedback paths introduced by the iterative waterfall model are shown in Figure 2.3. The feedback paths allow for correcting errors committed by a programmer during some phase, as and when these are detected in a later phase. For example, if during the testing phase a design error is identified, then the feedback path allows the design to be reworked and the changes to

be reflected in the design documents and all other subsequent documents. Please notice that in Figure 2.3 there is no feedback path to the feasibility stage. This is because once a team having accepted to take up a project, does not give up the project easily due to legal and moral reasons.

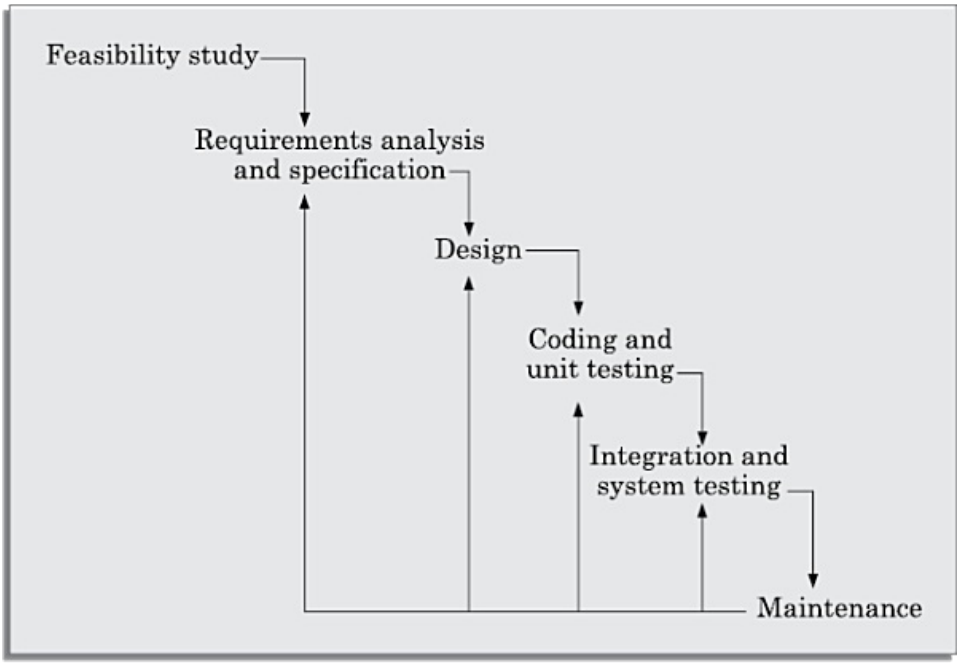

Figure 2.3: Iterative waterfall model.

Almost every life cycle model that we discuss are iterative in nature, except the classical waterfall model and the V-model—which are sequential in nature. In a sequential model, once a phase is complete, no work product of that phase are changed later.

Phase containment of errors

No matter how careful a programmer may be, he might end up committing some mistake or other while carrying out a life cycle activity. These mistakes result in errors (also called *faults* or *bugs*) in the work product. It is advantageous to detect these errors in the same phase in which they take place, since early detection of bugs reduces the effort and time required for correcting those. For example, if a design problem is detected in the design phase itself, then the problem can be taken care of much more easily than if the error is identified, say, at the end of the testing phase. In the later case, it would be necessary not only to rework the design, but also to appropriately redo the relevant coding as well as the testing activities, thereby incurring higher cost. It may not always be possible to detect all the errors in the same phase in which they are made. Nevertheless, the errors should be detected as

early as possible.

> The principle of detecting errors as close to their points of commitment as possible is known as *phase containment of errors*.

For achieving phase containment of errors, how can the developers detect almost all error that they commit in the same phase? After all, the end product of many phases are text or graphical documents, e.g. SRS document, design document, test plan document, etc. A popular technique is to rigorously review the documents produced at the end of a phase.

Phase overlap

Even though the strict waterfall model envisages sharp transitions to occur from one phase to the next (see Figure 2.3), in practice the activities of different phases overlap (as shown in Figure 2.4) due to two main reasons:

- In spite of the best effort to detect errors in the same phase in which they are committed, some errors escape detection and are detected in a later phase. These subsequently detected errors cause the activities of some already completed phases to be reworked. If we consider such rework after a phase is complete, we can say that the activities pertaining to a phase do not end at the completion of the phase, but overlap with other phases as shown in Figure 2.4.
- An important reason for phase overlap is that usually the work required to be carried out in a phase is divided among the team members. Some members may complete their part of the work earlier than other members. If strict phase transitions are maintained, then the team members who complete their work early would idle waiting for the phase to be complete, and are said to be in a *blocking state*. Thus the developers who complete early would idle while waiting for their team mates to complete their assigned work. Clearly this is a cause for wastage of resources and a source of cost escalation and inefficiency. As a result, in real projects, the phases are allowed to overlap. That is, once a developer completes his work assignment for a phase, proceeds to start the work for the next phase, without waiting for all his team members to complete their respective work allocations.

Considering these situations, the effort distribution for different phases with

time would be as shown in Figure 2.4.

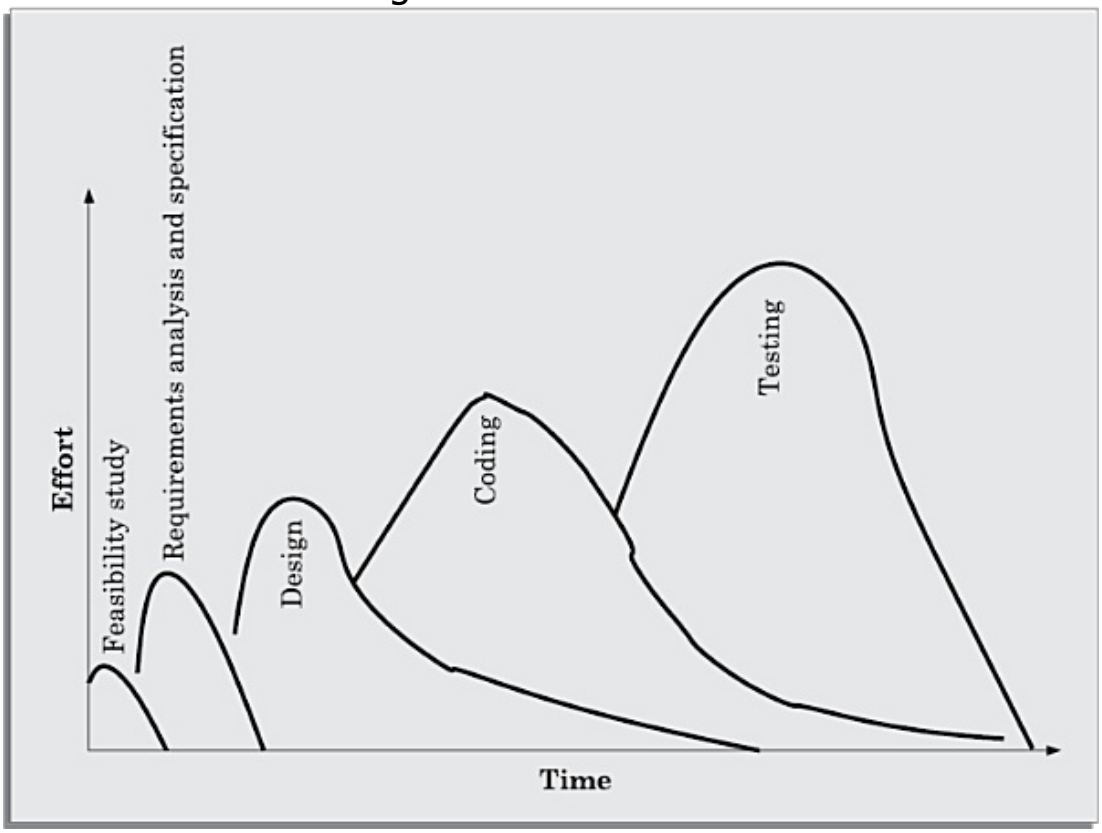

Figure 2.4: Distribution of effort for various phases in the iterative waterfall model.

Shortcomings of the iterative waterfall model

The iterative waterfall model is a simple and intuitive software development model. It was used satisfactorily during 1970s and 1980s. However, the characteristics of software development projects have changed drastically over years. In the 1970s and 1960s, software development projects spanned several years and mostly involved generic software product development. The projects are now shorter, and involve Customised software development. Further, software was earlier developed from scratch. Now the emphasis is on as much reuse of code and other project artifacts as possible. Waterfall-based models have worked satisfactorily over last many years in the past. The situation has changed substantially now. As pointed out in the first chapter several decades back, every software was developed from scratch. Now, not only software has become very large and complex, very few (if at all any) software project is being developed from scratch. The software services (customised software) are poised to become the dominant types of projects. In the present software development projects, use of waterfall model causes several problems. In this

context, the agile models have been proposed about a decade back that attempt to overcome the important shortcomings of the waterfall model by suggesting certain radical modification to the waterfall style of software development. In Section 2.4, we discuss the agile model. Some of the glaring shortcomings of the waterfall model when used in the present-day software development projects are as following:

Difficult to accommodate change requests: A major problem with the waterfall model is that the requirements need to be frozen before the development starts. Based on the frozen requirements, detailed plans are made for the activities to be carried out during the design, coding, and testing phases. Since activities are planned for the entire duration, substantial effort and resources are invested in the activities as developing the complete requirements specification, design for the complete functionality and so on. Therefore, accommodating even small change requests after the development activities are underway not only requires overhauling the plan, but also the artifacts that have already been developed.

Once requirements have been frozen, the waterfall model provides no scope for any modifications to the requirements.

While the waterfall model is inflexible to later changes to the requirements, evidence gathered from several projects points to the fact that later changes to requirements are almost inevitable. Even for projects with highly experienced professionals at all levels, as well as computer savvy customers, requirements are often missed as well as misinterpreted. Unless change requests are encouraged, the developed functionalities would be misfit to the true customer requirements. Requirement changes can arise due to a variety of reasons including the following—requirements were not clear to the customer, requirements were misunderstood, business process of the customer may have changed after the SRS document was signed off, etc. In fact, customers get clearer understanding of their requirements only after working on a fully developed and installed system.

The basic assumption made in the iterative waterfall model that methodical requirements gathering and analysis alone would comprehensively and correctly identify all the requirements by the end of the requirements phase is flawed.

Incremental delivery not supported: In the iterative waterfall model, the full software is completely developed and tested before it is delivered to the customer. There is no provision for any intermediate deliveries to occur. This

is problematic because the complete application may take several months or years to be completed and delivered to the customer. By the time the software is delivered, installed, and becomes ready for use, the customer's business process might have changed substantially. This makes the developed application a poor fit to the customer's requirements.

Phase overlap not supported: For most real life projects, it becomes difficult to follow the rigid phase sequence prescribed by the waterfall model. By the term *a rigid phase sequence*, we mean that a phase can start only after the previous phase is complete in all respects. As already discussed, strict adherence to the waterfall model creates *blocking states*. The waterfall model is usually adapted for use in real-life projects by allowing overlapping of various phases as shown in Figure 2.4.

Error correction unduly expensive: In waterfall model, validation is delayed till the complete development of the software. As a result, the defects that are noticed at the time of validation incur expensive rework and result in cost escalation and delayed delivery.

Limited customer interactions: This model supports very limited customer interactions. It is generally accepted that software developed in isolation from the customer is the cause of many problems. In fact, interactions occur only at the start of the project and at project completion. As a result, the developed software usually turns out to be a misfit to the customer's actual requirements.

Heavy weight: The waterfall model overemphasises documentation. A significant portion of the time of the developers is spent in preparing documents, and revising them as changes occur over the life cycle. Heavy documentation though useful during maintenance and for carrying out review, is a source of team inefficiency.

No support for risk handling and code reuse: It becomes difficult to use the waterfall model in projects that are susceptible to various types of risks, or those involving significant reuse of existing development artifacts. Please recollect that software services types of projects usually involve significant reuse.

2.2.3 V-Model

A popular development process model, V-model is a variant of the waterfall model. As is the case with the waterfall model, this model gets its name from its visual appearance (see Figure 2.5). In this model verification and

validation activities are carried out throughout the development life cycle, and therefore the chances bugs in the work products considerably reduce. This model is therefore generally considered to be suitable for use in projects concerned with development of safety-critical software that are required to have high reliability.

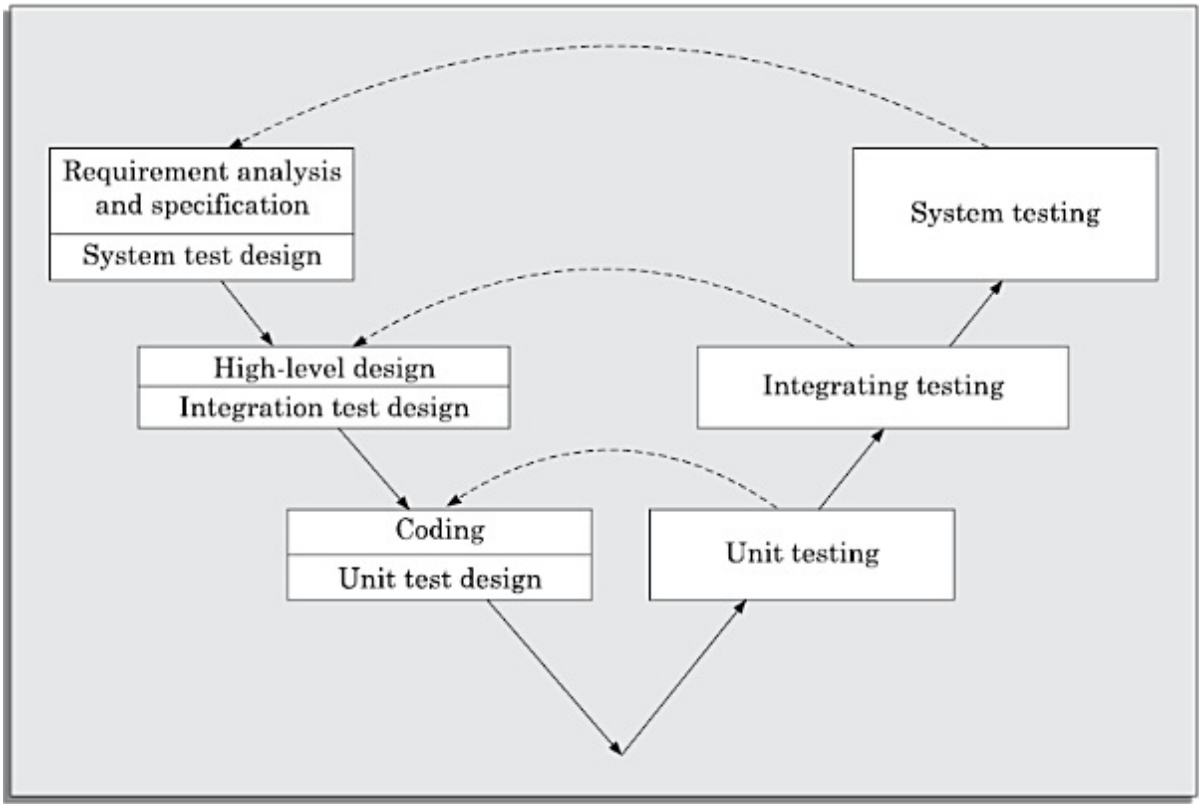

Figure 2.5: V-model.

As shown in Figure 2.5, there are two main phases—development and validation phases. The left half of the model comprises the development phases and the right half comprises the validation phases.

- In each development phase, along with the development of a work product, test case design and the plan for testing the work product are carried out, whereas the actual testing is carried out in the validation phase. This validation plan created during the development phases is carried out in the corresponding validation phase which have been shown by dotted arcs in Figure 2.5.
- In the validation phase, testing is carried out in three steps—unit, integration, and system testing. The purpose of these three different steps of testing during the validation phase is to detect defects that arise in the corresponding phases of software development—

requirements analysis and specification, design, and coding respectively.

V-model *versus* waterfall model

We have already pointed out that the V-model can be considered to be an extension of the waterfall model. However, there are major differences between the two. As already mentioned, in contrast to the iterative waterfall model where testing activities are confined to the testing phase only, in the V-model testing activities are spread over the entire life cycle. As shown in Figure 2.5, during the requirements specification phase, the system test suite design activity takes place. During the design phase, the integration test cases are designed. During coding, the unit test cases are designed. Thus, we can say that in this model, development and validation activities proceed hand in hand.

Advantages of V-model

The important advantages of the V-model over the iterative waterfall model are as following:

- In the V-model, much of the testing activities (test case design, test planning, etc.) are carried out in parallel with the development activities. Therefore, before testing phase starts significant part of the testing activities, including test case design and test planning, is already complete. Therefore, this model usually leads to a shorter testing phase and an overall faster product development as compared to the iterative model.
- Since test cases are designed when the schedule pressure has not built up, the quality of the test cases are usually better.
- The test team is reasonably kept occupied throughout the development cycle in contrast to the waterfall model where the testers are active only during the testing phase. This leads to more efficient manpower utilisation.
- In the V-model, the test team is associated with the project from the beginning. Therefore they build up a good understanding of the development artifacts, and this in turn, helps them to carry out effective testing of the software. In contrast, in the waterfall model often the test team comes on board late in the development cycle,

since no testing activities are carried out before the start of the implementation and testing phase.

Disadvantages of V-model

Being a derivative of the classical waterfall model, this model inherits most of the weaknesses of the waterfall model.

2.2.4 Prototyping Model

The prototype model is also a popular life cycle model. The prototyping model can be considered to be an extension of the waterfall model. This model suggests building a working *prototype* of the system, before development of the actual software. A prototype is a toy and crude implementation of a system. It has limited functional capabilities, low reliability, o r inefficient performance as compared to the actual software. A prototype can be built very quickly by using several shortcuts. The shortcuts usually involve developing inefficient, inaccurate, or dummy functions. The shortcut implementation of a function, for example, may produce the desired results by using a table look-up rather than by performing the actual computations. Normally the term *rapid prototyping* is used when software tools are used for prototype construction. For example, tools based on *fourth generation languages* (4GL) may be used to construct the prototype for the GUI parts.

Necessity of the prototyping model

The prototyping model is advantageous to use for specific types of projects. In the following, we identify three types of projects for which the prototyping model can be followed to advantage:

- It is advantageous to use the prototyping model for development of the *graphical user interface* (GUI) part of an application. Through the use of a prototype, it becomes easier to illustrate the input data formats, messages, reports, and the interactive dialogs to the customer. This is a valuable mechanism for gaining better understanding of the customers' needs. In this regard, the prototype model turns out to be especially useful in developing the *graphical user interface* (GUI) part of a system. For the user, it becomes much easier

to form an opinion regarding what would be more suitable by experimenting with a working user interface, rather than trying to imagine the working of a hypothetical user interface.

The GUI part of a software system is almost always developed using the prototyping model.

- The prototyping model is especially useful when the exact technical solutions are unclear to the development team. A prototype can help them to critically examine the technical issues associated with product development. For example, consider a situation where the development team has to write a command language interpreter as part of a graphical user interface development. Suppose none of the team members has ever written a compiler before. Then, this lack of familiarity with a required development technology is a technical risk. This risk can be resolved by developing a prototype compiler for a very small language to understand the issues associated with writing a compiler for a command language. Once they feel confident in writing compiler for the small language, they can use this knowledge to develop the compiler for the command language. Often, major design decisions depend on issues such as the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype is often the best way to resolve the technical issues.
- An important reason for developing a prototype is that it is impossible to "get it right" the first time. As advocated by Brooks [1975], one must plan to throw away the software in order to develop a good software later. Thus, the prototyping model can be deployed when development of highly optimised and efficient software is required.

From the above discussions, we can conclude the following:

The prototyping model is considered to be useful for the development of not only the GUI parts of a software, but also for a software project for which certain technical issues are not clear to the development team.

Life cycle activities of prototyping model

The prototyping model of software development is graphically shown in

Figure 2.6. As shown in Figure 2.6, software is developed through two major activities—prototype construction and iterative waterfall-based software development.

Prototype development: Prototype development starts with an initial requirements gathering phase. A quick design is carried out and a prototype is built. The developed prototype is submitted to the customer for evaluation. Based on the customer feedback, the requirements are refined and the prototype is suitably modified. This cycle of obtaining customer feedback and modifying the prototype continues till the customer approves the prototype.

Iterative development: Once the customer approves the prototype, the actual software is developed using the iterative waterfall approach. In spite of the availability of a working prototype, the SRS document is usually needed to be developed since the SRS document is invaluable for carrying out traceability analysis, verification, and test case design during later phases. However, for GUI parts, the requirements analysis and specification phase becomes redundant since the working prototype that has been approved by the customer serves as an animated requirements specification.

The code for the prototype is usually thrown away. However, the experience gathered from developing the prototype helps a great deal in developing the actual system.

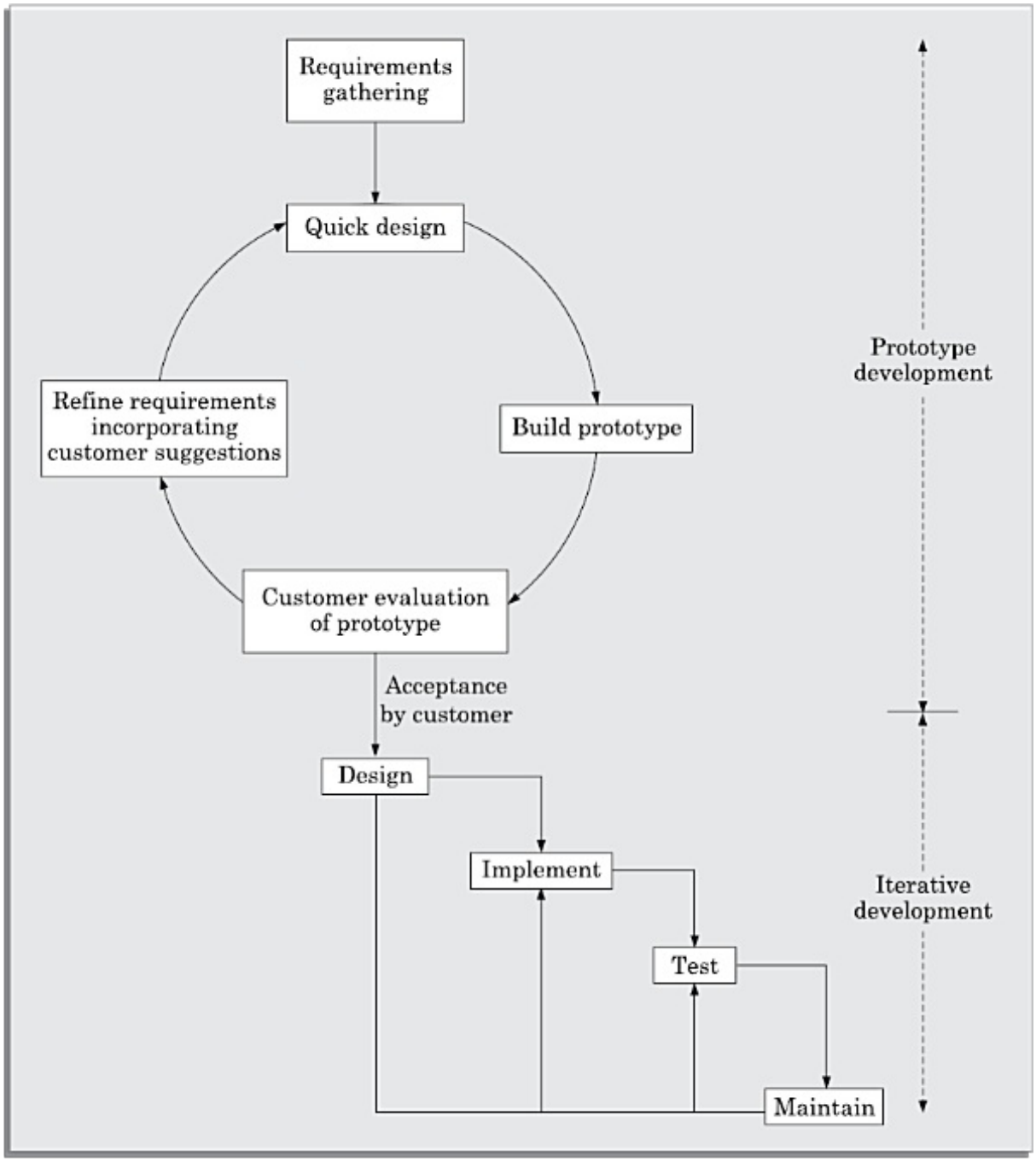

Figure 2.6: Prototyping model of software development.

Even though the construction o f a throwaway prototype might involve incurring additional cost, for systems with unclear customer requirements and for systems with unresolved technical issues, the overall development cost usually turns out to be lower compared to an equivalent system developed using the iterative waterfall model.

By constructing the prototype and submitting it for user evaluation, many customer requirements get properly defined and technical issues get resolved by experimenting with the prototype. This minimises later change requests

from the customer and the associated redesign costs.

Strengths of the prototyping model

This model is the most appropriate for projects that suffer from technical and requirements risks. A constructed prototype helps overcome these risks.

Weaknesses of the prototyping model

The prototype model can increase the cost of development for projects that are routine development work and do not suffer from any significant risks. Even when a project is susceptible to risks, the prototyping model is effective only for those projects for which the risks can be identified upfront before the development starts. Since the prototype is constructed only at the start of the project, the prototyping model is ineffective for risks identified later during the development cycle. The prototyping model would not be appropriate for projects for which the risks can only be identified after the development is underway.

2.2.5 Incremental Development Model

This life cycle model is sometimes referred to as the *successive versions* model and sometimes as the incremental model. In this life cycle model, first a simple working system implementing only a few basic features is built and delivered to the customer. Over many successive iterations successive versions are implemented and delivered to the customer until the desired system is realised. The incremental development model has been shown in Figure 2.7.

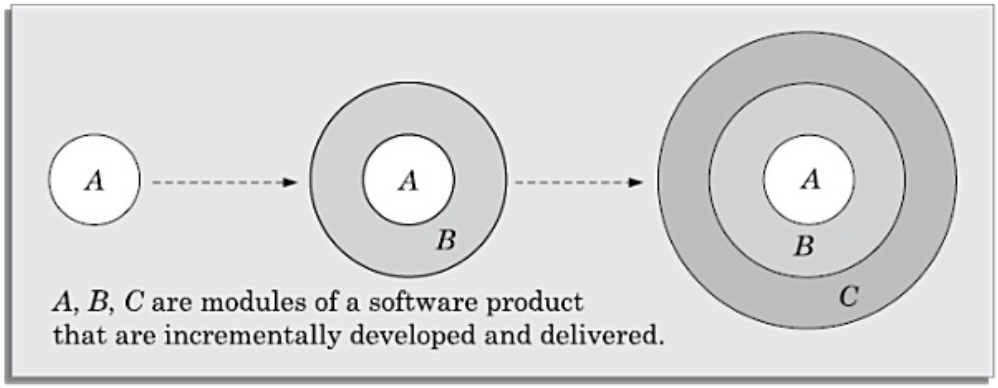

Figure 2.7: Incremental software development.

Life cycle activities of incremental development model

In the incremental life cycle model, the requirements of the software are first broken down into several modules or features that can be incrementally constructed and delivered. This has been pictorially depicted in Figure 2.7. At any time, plan is made only for the next increment and no long-term plans are made. Therefore, it becomes easier to accommodate change requests from the customers.

The development team first undertakes to develop the core features of the system. The core or basic features are those that do not need to invoke any services from the other features. On the other hand, non-core features need services from the core features. Once the initial core features are developed, these are refined into increasing levels of capability by adding new functionalities in successive versions. Each incremental version is usually developed using an iterative waterfall model of development. The incremental model is schematically shown in Figure 2.8. As each successive version of the software is constructed and delivered to the customer, the customer feedback is obtained on the delivered version and these feedbacks are incorporated in the next version. Each delivered version of the software incorporates additional features over the previous version and also refines the features that were already delivered to the customer.

The incremental model has schematically been shown in Figure 2.8. After the requirements gathering and specification, the requirements are split into several versions. Starting with the core (version 1), in each successive increment, the next version is constructed using an iterative waterfall model of development and deployed at the customer site. After the last (shown as version n) has been developed and deployed at the client site, the full software is deployed.

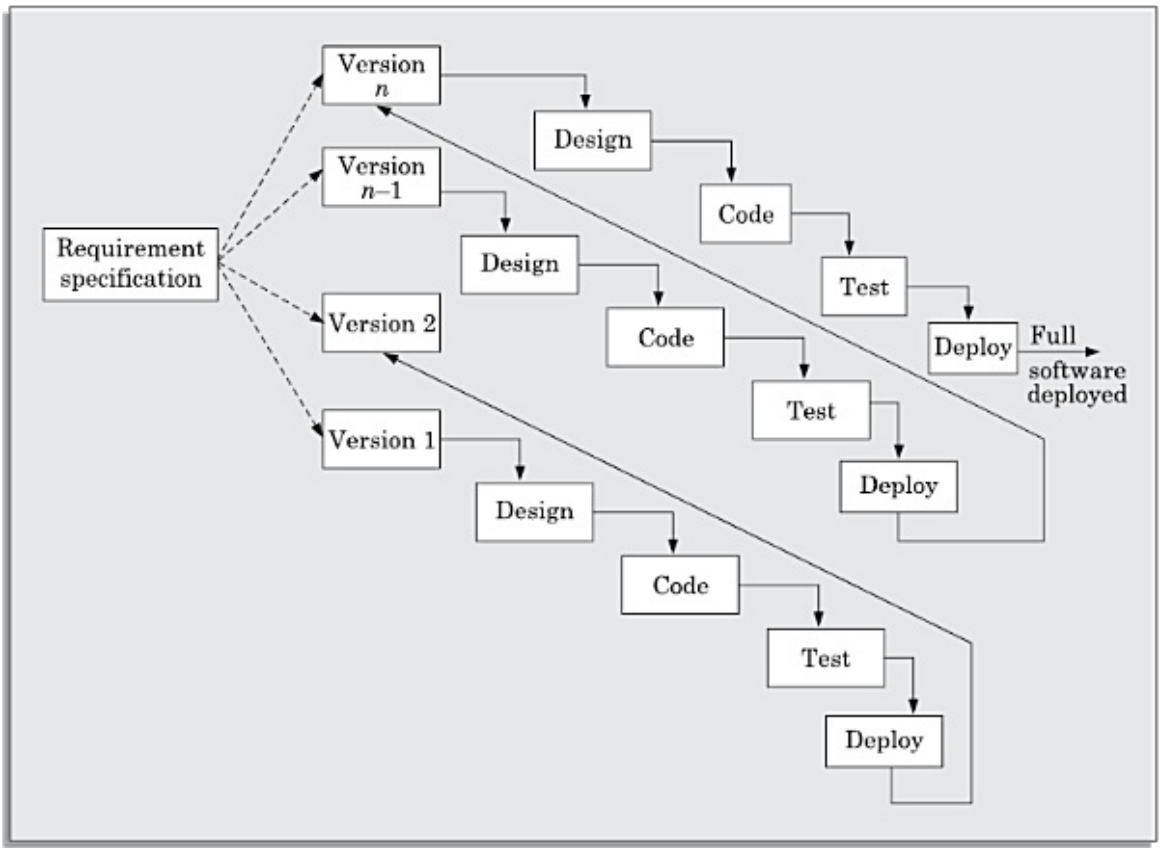

Figure 2.8: Incremental model of software development.

Advantages

The incremental development model offers several advantages. Two important ones are the following:

- **Error reduction:** The core modules are used by the customer from the beginning and therefore these get tested thoroughly. This reduces chances of errors in the core modules of the final product, leading to greater reliability of the software.
- **Incremental resource deployment:** This model obviates the need for the customer to commit large resources at one go for development of the system. It also saves the developing organisation from deploying large resources and manpower for a project in one go.

2.2.6 Evolutionary Model

This model has many of the features of the incremental model. As in case of the incremental model, the software is developed over a number of increments. At each increment, a concept (feature) is implemented and is deployed at the client site. The software is

successively refined and feature-enriched until the full software is realised. The principal idea behind the evolutionary life cycle model is conveyed by its name. In the incremental development model, complete requirements are first developed and the SRS document prepared. In contrast, in the evolutionary model, the requirements, plan, estimates, and solution evolve over the iterations, rather than fully defined and frozen in a major up-front specification effort before the development iterations begin. Such evolution is consistent with the pattern of unpredictable feature discovery and feature changes that take place in new product development.

Though the evolutionary model can also be viewed as an extension of the waterfall model, but it incorporates a major paradigm shift that has been widely adopted in many recent life cycle models. Due to obvious reasons, the evolutionary software development process is sometimes referred to as *design a little, build a little, test a little, deploy a little* model. This means that after the requirements have been specified, the design, build, test, and deployment activities are iterated. A schematic representation of the evolutionary model of development has been shown in Figure 2.9.

Advantages

The evolutionary model of development has several advantages. Two important advantages of using this model are the following:

- **Effective elicitation of actual customer requirements:** In this model, the user gets a chance to experiment with a partially developed software much before the complete requirements are developed. Therefore, the evolutionary model helps to accurately elicit user requirements with the help of feedback obtained on the delivery of different versions of the software. As a result, the change requests after delivery of the complete software gets substantially reduced.
- **Easy handling change requests:** In this model, handling change requests is easier as no long term plans are made. Consequently, reworks required due to change requests are normally much smaller compared to the sequential models.

Disadvantages

The main disadvantages of the successive versions model are as follows:

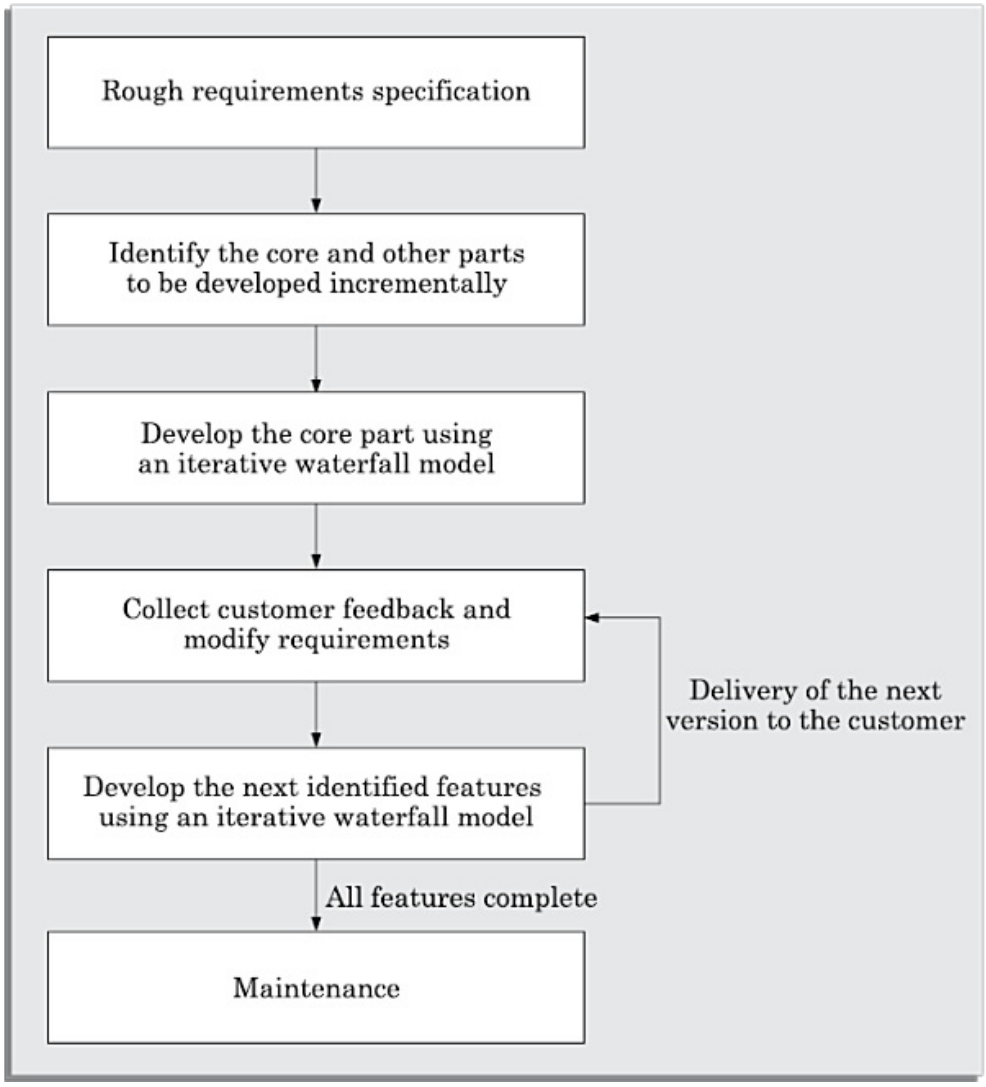

Figure 2.9: Evolutionary model of software development.

- **Feature division into incremental parts can be non-trivial:** For many development projects, especially for small-sized projects, it is difficult to divide the required features into several parts that can be incrementally implemented and delivered. Further, even for larger problems, often the features are so interwined and dependent on each other that even an expert would need considerable effort to plan the incremental deliveries.
- ***Ad hoc*** **design:** Since at a time design for only the current increment is done, the design can become ad hoc without specific attention being paid to maintainability and optimality. Obviously, for moderate sized problems and for those for which the customer requirements are clear, the iterative waterfall model can yield a better solution.

Applicability of the evolutionary model

The evolutionary model is normally useful for very large products, where it is easier to find modules for incremental implementation. Often evolutionary model is used when the customer prefers to receive the product in increments so that he can start using the different features as and when they are delivered rather than waiting all the time for the full product to be developed and delivered. Another important category of projects for which the evolutionary model is suitable, is projects using object-oriented development.

The evolutionary model is well-suited to use in object-oriented software development projects.

Evolutionary model is appropriate for object-oriented development project, since it is easy to partition the software into stand alone units in terms of the classes. Also, classes are more or less self contained units that can be developed independently.

2.3 RAPID APPLICATION DEVELOPMENT (RAD)

The *rapid application development* (RAD) model was proposed in the early nineties in an attempt to overcome the rigidity of the waterfall model (and its derivatives) that makes it difficult to accommodate any change requests from the customer. It proposed a few radical extensions to the waterfall model. This model has the features of both prototyping and evolutionary models. It deploys an evolutionary delivery model to obtain and incorporate the customer feedbacks on incrementally delivered versions.

In this model prototypes are constructed, and incrementally the features are developed and delivered to the customer. But unlike the prototyping model, the prototypes are not thrown away but are enhanced and used in the software construction

The major goals of the RAD model are as follows:

- To decrease the time taken and the cost incurred to develop software systems.
- To limit the costs of accommodating change requests.
- To reduce the communication gap between the customer and the developers.

Main motivation

In the iterative waterfall model, the customer requirements need to be gathered, analysed, documented, and signed off upfront, before any development could start. However, often clients do not know what they exactly wanted until they saw a working system. It has now become well accepted among the practitioners that only through the process commenting on an installed application that the exact requirements can be brought out. But in the iterative waterfall model, the customers do not get to see the software, until the development is complete in all respects and the software has been delivered and installed. Naturally, the delivered software often does not meet the customer expectations and many change request are generated by the customer. The changes are incorporated through subsequent maintenance efforts. This made the cost of accommodating the changes extremely high and it usually took a long time to have a good solution in place that could reasonably meet the requirements of the customers. The RAD model tries to overcome this problem by inviting and incorporating customer feedback on successively developed and refined prototypes.

2.3.1 Working of RAD

In the RAD model, development takes place in a series of short cycles or iterations. At any time, the development team focuses on the present iteration only, and therefore plans are made for one increment at a time. The time planned for each iteration is called a *time box*. Each iteration is planned to enhance the implemented functionality of the application by only a small amount. During each time box, a quick-and-dirty prototype-style software for some functionality is developed. The customer evaluates the prototype and gives feedback on the specific improvements that may be necessary. The prototype is refined based on the customer feedback. Please note that the prototype is not meant to be released to the customer for regular use though.

The development team almost always includes a customer representative to clarify the requirements. This is intended to make the system tuned to the exact customer requirements and also to bridge the communication gap between the customer and the development team. The development team usually consists of about five to six members, including a customer representative.

How does RAD facilitate accommodation of change requests?

The customers usually suggest changes to a specific feature only after they have used it. Since the features are delivered in small increments, the customers are able to give their change requests pertaining to a feature already delivered. Incorporation of such change requests just after the delivery of an incremental feature saves cost as this is carried out before large investments have been made in development and testing of a large number of features.

How does RAD facilitate faster development?

The decrease in development time and cost, and at the same time an increased flexibility to incorporate changes are achieved in the RAD model in two main ways—minimal use of planning and heavy reuse of any existing code through rapid prototyping. The lack of long-term and detailed planning gives the flexibility to accommodate later requirements changes. Reuse of existing code has been adopted as an important mechanism of reducing the development cost.

RAD model emphasises code reuse as an important means for completing a project faster. In fact, the adopters of the RAD model were the earliest to embrace object-oriented languages and practices. Further, RAD advocates use of specialised tools to facilitate fast creation of working prototypes. These specialised tools usually support the following features:

- Visual style of development.
- Use of reusable components.

2.3.2 Applicability of RAD Model

The following are some of the characteristics of an application that indicate its suitability to RAD style of development:

- **Customised software:** As already pointed out a customised software is developed for one or two customers only by adapting an existing software. In customised software development projects, substantial reuse is usually made of code from pre-existing software. For example, a company might have developed a software for automating the data processing activities at one or more educational institutes. When any other institute requests for an automation package to be developed, typically only a few aspects needs to be tailored—since among different educational

institutes, most of the data processing activities such as student registration, grading, fee collection, estate management, accounting, maintenance of staff service records etc. are similar to a large extent. Projects involving such tailoring can be carried out speedily and cost-effectively using the RAD model.

- **Non-critical software:** The RAD model suggests that a quick and dirty software should first be developed and later this should be refined into the final software for delivery. Therefore, the developed product is usually far from being optimal in performance and reliability. In this regard, for well understood development projects and where the scope of reuse is rather restricted, the Iiterative waterfall model may provide a better solution.
- **Highly constrained pro ject schedule:** RAD aims to reduce development time at the expense of good documentation, performance, and reliability. Naturally, for projects with very aggressive time schedules, RAD model should be preferred.
- **Large software:** Only for software supporting many features (large software) can incremental development and delivery be meaningfully carried out.

Application characteristics that render RAD unsuitable

The RAD style of development is not advisable if a development project has one or more of the following characteristics:

- **Generic products (wide distribution):** As we have already pointed out in Chapter 1, software products are generic in nature and usually have wide distribution. For such systems, optimal performance and reliability are imperative in a competitive market. As it has already been discussed, the RAD model of development may not yield systems having optimal performance and reliability.
- **Requirement of optimal performance and/or reliability:** For certain categories of products, optimal performance or reliability is required. Examples of such systems include an operating system (high reliability required) and a flight simulator software (high performance required). If such systems are to be developed using the RAD model, the desired product performance and reliability may not be realised.
- **Lack of similar products:** If a company has not developed similar

software, then it would hardly be able to reuse much of the existing artifacts. In the absence of sufficient plug-in components, it becomes difficult to develop rapid prototypes through reuse, and use of RAD model becomes meaningless.

- **Monolithic entity:** For certain software, especially small-sized software, it may be hard to divide the required features into parts that can be incrementally developed and delivered. In this case, it becomes difficult to develop a software incrementally.

2.3.3 Comparison of RAD with Other Models

In this section, we compare the relative advantages and disadvantages of RAD with other life cycle models.

RAD *versus* prototyping model

In the prototyping model, the developed prototype is primarily used by the development team to gain insights into the problem, choose between alternatives, and elicit customer feedback. The code developed during prototype construction is usually thrown away. In contrast, in RAD it is the developed prototype that evolves into the deliverable software.

Though RAD is expected to lead to faster software development compared to the traditional models (such as the prototyping model), though the quality and reliability would be inferior.

RAD *versus* iterative waterfall model

In the iterative waterfall model, all the functionalities of a software are developed together. On the other hand, in the RAD model the product functionalities are developed incrementally through heavy code and design reuse. Further, in the RAD model customer feedback is obtained on the developed prototype after each iteration and based on this the prototype is refined. Thus, it becomes easy to accommodate any request for requirements changes. However, the iterative waterfall model does not support any mechanism to accommodate any requirement change requests. The iterative waterfall model does have some important advantages that include the following. Use of the iterative waterfall model leads to production of good quality documentation which can help during software maintenance. Also, the developed software usually has better quality and reliability than that

developed using RAD.

RAD *versus* evolutionary model

Incremental development is the hallmark of both evolutionary and RAD models. However, in RAD each increment results in essentially a quick and dirty prototype, whereas in the evolutionary model each increment is systematically developed using the iterative waterfall model. Also in the RAD model, software is developed in much shorter increments compared the evolutionary model. In other words, the incremental functionalities that are developed are of fairly larger granularity in the evolutionary model.

2.4 AGILE DEVELOPMENT MODELS

As already pointed out, though the iterative waterfall model has been very popular during the 1970s and 1980s, developers face several problems while using it on present day software projects. The main difficulties included handling change requests from customers during product development, and the unreasonably high cost and time that is incurred while developing customised applications. Capers Jones carried out research involving 800 real-life software development projects, and concluded that on the average 40 per cent of the requirements is arrived after the development has already begun. In this context, over the last two decade or so, several life cycle models have been proposed to overcome the important shortcomings of the waterfall-based models that become conspicuous when used in modern software development projects.

Over the last two decades or so, projects using iterative waterfall-based life cycle models are becoming rare due to the rapid shift in the characteristics of the software development projects over time. Two changes that are becoming noticeable are rapid shift from development of software products to development of customised software and the increased emphasis and scope for reuse.

In the following, a few reasons why the waterfall-based development was becoming difficult to use in project in recent times:

- In the traditional iterative waterfall-based software development models, the requirements for the system are determined at the start of a development project and are assumed to be fixed from that point on. Later changes to the requirements after the SRS document has been

completed are discouraged. If at all any later requirement changes becomes unavoidable, then the cost of accommodating it becomes prohibitively high. On the other hand, accumulated experience indicates that customers frequently change their requirements during the development period due to a variety of reasons.

- As pointed out in Chapter 1, over the last two decades or so, customised applications (services) has become common place and the sales revenue generated world wide from services already exceeds that of the software products. Clearly, iterative waterfall model is not suitable for development of such software. Since customization essentially involves reusing most of the parts of an existing application and consists of only carrying out minor modifications by writing minimal amounts of code. For such development projects, the need for more appropriate development models was deeply felt, and many researchers started to investigate in this direction.
- Waterfall model is called a *heavy weight* model, since there is too much emphasis on producing documentation and usage of tools. This is often a source of inefficiency and causes the project completion time to be much longer in comparison to the customer expectations.
- Waterfall model prescribes almost no customer interactions after the requirements have been specified. In fact, in the waterfall model of software development, customer interactions are largely confined to the project initiation and project completion stages.

The agile software development model was proposed in the mid-1990s to overcome the serious shortcomings of the waterfall model of development identified above. The agile model was primarily designed to help a project to adapt to change requests quickly.[1]Thus, a major aim of the agile models is to facilitate quick project completion. But, how is agility achieved in these models? Agility is achieved by fitting the process to the project, i.e. removing activities that may not be necessary for a specific project. Also, anything that that wastes time and effort is avoided.

Please note that agile model is being used as an umbrella term to refer to a group of development processes. These processes share certain common characteristics, but do have certain subtle differences among themselves. A few popular agile SDLC models are the following:

- Crystal

- Atern (formerly DSDM)
- Feature-driven development
- Scrum
- Extreme programming (XP)
- Lean development
- Unified process

In the agile model, the requirements are decomposed into many small parts that can be incrementally developed. The agile model adopts an iterative approach. Each incremental part is developed over an iteration. Each iteration is intended to be small and easily manageable and lasting for a couple of weeks only. At a time, only one increment is planned, developed, and then deployed at the customer site. No long-term plans are made. The time to complete an iteration is called a *time box*. The implication of the term *time box* is that the end date for an iteration does not change. That is, the delivery date is considered sacrosanct. The development team can, however, decide to reduce the delivered functionality during a time box if necessary.

A central principle of the agile model is the delivery of an increment to the customer after each time box. A few other principles that are central to the agile model are discussed below.

2.4.1 Essential Idea behind Agile Models

For establishing close contact with the customer during development and to gain a clear understanding of the domain-specific issues, each agile project usually includes a customer representative in the team. At the end of each iteration, stakeholders and the customer representative review the progress made and re-evaluate the requirements. A distinguishing characteristics of the agile models is frequent delivery of software increments to the customer.

Agile model emphasise face-to-face communication over written documents. It is recommended that the development team size be deliberately kept small (5–9 people) to help the team members meaningfully engage in face-to-face communication and have collaborative work environment. It is implicit then that the agile model is suited to the development of small projects. However, if a large project is required to be developed using the agile model, it is likely that the collaborating teams might work at different locations. In this case, the different teams are needed to maintain as much daily contact as possible through video conferencing,

telephone, e-mail, etc.

> The agile model emphasises incremental release of working software as the primary measure of progress.

The following important principles behind the agile model were publicised in the agile manifesto in 2001:

- Working software over comprehensive documentation.
- Frequent delivery of incremental versions of the software to the customer in intervals of few weeks.
- Requirement change requests from the customer are encouraged and are efficiently incorporated.
- Having competent team members and enhancing interactions among them is considered much more important than issues such as usage of sophisticated tools or strict adherence to a documented process. It is advocated that enhanced communication among the development team members can be realised through face-to-face communication rather than through exchange of formal documents.
- Continuous interaction with the customer is considered much more important rather than effective contract negotiation. A customer representatives is required to be a part of the development team, thus facilitating close, daily co-operation between customers and developers.

Agile development projects usually deploy pair programming.

> In pair programming, two programmers work together at one work station. One types in code while the other reviews the code as it is typed in. The two programmers switch their roles every hour or so.

Several studies indicate that programmers working in pairs produce compact well-written programs and commit fewer errors as compared to programmers working alone.

Advantages and disadvantages of agile methods

The agile methods derive much of their agility by relying on the tacit knowledge of the team members about the development project and informal communications to clarify issues, rather than spending significant amounts of time in preparing formal documents and

reviewing them. Though this eliminates some overhead, but lack of adequate documentation may lead to several types of problems, which are as follows:

- Lack of formal documents leaves scope for confusion and important decisions taken during different phases can be misinterpreted at later points of time by different team members.
- In the absence of any formal documents, it becomes difficult to get important project decisions such as design decisions to be reviewed by external experts.
- When the project completes and the developers disperse, maintenance can become a problem.

2.4.2 Agile *versus* Other Models

In the following subsections, we compare the characteristics of the agile model with other models of development.

Agile model *versus* iterative waterfall model

The waterfall model is highly structured and systematically steps through requirements-capture, analysis, specification, design, coding, and testing stages in a planned sequence. Progress is generally measured in terms of the number of completed and reviewed artifacts such as requirement specifications, design documents, test plans, code reviews, etc. for which review is complete. In contrast, while using an agile model, progress is measured in terms of the developed and delivered functionalities. In agile model, delivery of working versions of a software is made in several increments. However, as regards to similarity it can be said that agile teams use the waterfall model on a small scale, repeating the entire waterfall cycle in every iteration.

If a project being developed using waterfall model is cancelled mid-way during development, then there is nothing to show from the abandoned project beyond several documents. With agile model, even if a project is cancelled midway, it still leaves the customer with some worthwhile code, that might possibly have already been put into live operation.

Agile *versus* exploratory programming

Though a few similarities do exist between the agile and exploratory programming styles, there are vast differences between the two as

well. Agile development model's frequent re- evaluation of plans, emphasis on face-to-face communication, and relatively sparse use of documentation are similar to that of the exploratory style. Agile teams, however, do follow defined and disciplined processes and carry out systematic requirements capture, rigorous designs, compared to chaotic coding in exploratory programming.

Agile model *versus* RAD model

The important differences between the agile and the RAD models are the following:

- Agile model does not recommend developing prototypes, but emphasises systematic development of each incremental feature. In contrast, the central theme of RAD is based on designing quick-and-dirty prototypes, which are then refined into production quality code.
- Agile projects logically break down the solution into features that are incrementally developed and delivered. The RAD approach does not recommend this. Instead, developers using the RAD model focus on developing all the features of an application by first doing it badly and then successively improving the code over time.
- Agile teams only demonstrate completed work to the customer. In contrast, RAD teams demonstrate to customers screen mock ups, and prototypes, that may be based on simplifications such as table look-ups rather than actual computations.

2.4.3 Extreme Programming Model

Extreme programming (XP) is an important process model under the agile umbrella and was proposed by Kent Beck in 1999. The name of this model reflects the fact that it recommends taking these *best practices* that have worked well in the past in program development projects to extreme levels. This model is based on a rather simple philosophy: "If something is known to be beneficial, why not put it to constant use?" Based on this principle, it puts forward several key practices that need to be practised to the extreme. Please note that most of the key practices that it emphasises were already recognised as good practices for some time.

Good practices that need to be practised to the extreme

In the following subsections, we mention some of the good practices that have been recognised in the extreme programming model and the suggested way to maximise their use:

Code review: It is good since it helps detect and correct problems most efficiently. It suggests *pair programming* as the way to achieve continuous review. In pair programming, coding is carried out by pairs of programmers. The programmers take turn in writing programs and while one writes the other reviews code that is being written.

Testing: Testing code helps to remove bugs and improves its reliability. XP suggests *test-driven development* (TDD) to continually write and execute test cases. In the TDD approach, test cases are written even before any code is written.

Incremental development: Incremental development is good, since it helps to get customer feedback, and extent of features delivered is a reliable indicator of progress. It suggests that the team should come up with new increments every few days.

Simplicity: Simplicity makes it easier to develop good quality code, as well as to test and debug it. Therefore, one should try to create the simplest code that makes the basic functionality being written to work. For creating the simplest code, one can ignore the aspects such as efficiency, reliability, maintainability, etc. Once the simplest thing works, other aspects can be introduced through refactoring.

Design: Since having a good quality design is important to develop a good quality solution, everybody should design daily. This can be achieved through *refactoring*, whereby a working code is improved for efficiency and maintainability.

Integration testing: It is important since it helps identify the bugs at the interfaces of different functionalities. To this end, extreme programming suggests that the developers should achieve continuous integration, by building and performing integration testing several times a day.

Basic idea of extreme programming model

XP is based on frequent releases (called *iteration*), during which the developers implement "user stories". User stories are similar to use cases, but are more informal and are simpler. A user story is the conversational description by the user about a feature of the required

system. For example, a user story about a library software can be:

- A library member can issue a book.
- A library member can query about the availability of a book.
- A library member should be able to return a borrowed book.

A user story is a simplistic statement of a customer about a functionality he needs, it does not mention about finer details such as the different scenarios that can occur, the precondition (state at which the system) to be satisfied before the feature can be invoked, etc.

On the basis of user stories, the project team proposes "metaphors"—a common vision of how the system would work. The development team may decide to construct a *spike* for some feature. A *spike*, is a very simple program that is constructed to explore the suitability of a solution being proposed. A spike can be considered to be similar to a prototype.

X P prescribes several basic activities to be part of the software development process. We discuss these activities in the following subsections:

Coding: XP argues that code is the crucial part of any system development process, since without code it is not possible to have a working system. Therefore, utmost care and attention need to be placed on coding activity. However, the concept of code as used in XP has a slightly different meaning from what is traditionally understood. For example, coding activity includes drawing diagrams (modelling) that will be transformed to code, scripting a web-based system, and choosing among several alternative solutions.

Testing: XP places high importance on testing and considers it be the primary means for developing a fault-free software.

Listening: The developers need to carefully listen to the customers if they have to develop a good quality software. Programmers may not necessarily be having an in-depth knowledge of the the specific domain of the system under development. On the other hand, customers usually have this domain knowledge. Therefore, for the programmers to properly understand what the functionality of the system should be, they have to listen to the customer.

Designing: Without proper design, a system implementation becomes too complex and the dependencies within the system become too numerous and it becomes very difficult to comprehend the solution, and thereby making maintenance prohibitively expensive. A good design should result in

elimination of complex dependencies within a system. Thus, effective use of a suitable design technique is emphasised.

Feedback: It espouses the wisdom: "A system staying out of users is trouble waiting to happen". It recognises the importance of user feedback in understanding the exact customer requirements. The time that elapses between the development of a version and collection of feedback on it is critical to learning and making changes. It argues that frequent contact with the customer makes the development effective.

Simplicity: A corner-stone of XP is based on the principle: "build something simple that will work today, rather than trying to build something that would take time and yet may never be used". This in essence means that attention should be focused on specific features that are immediately needed and making them work, rather than devoting time and energy on speculations about future requirements.

XP is in favour of making the solution to a problem as simple as possible. In contrast, the traditional system development methods recommend planning for reusability and future extensibility of code and design at the expense of higher code and design complexity.

Applicability of extreme programming model

The following are some of the project characteristics that indicate the suitability of a project for development using extreme programming model:

Projects involving new technology or research pro jects: In this case, the requirements change rapidly and unforeseen technical problems need to be resolved.

Small projects: Extreme programming was proposed in the context of small teams as face to face meeting is easier to achieve.

Project characteristics not suited to development using agile models

The following are some of the project characteristics that indicate unsuitability of agile development model for use in a development project:

- **Stable requirements:** Conventional development models are more suited to use in projects characterised by stable requirements. For such

projects, it is known that few changes, if at all, will occur. Therefore, process models such as iterative waterfall model that involve making long-term plans during project initiation can meaningfully be used.

- **Mission critical or safety critical systems:** In the development of such systems, the traditional SDLC models are usually preferred to ensure reliability.

2.4.4 Scrum Model

In the scrum model, a project is divided into small parts of work that can be incrementally developed and delivered over time boxes that are called *sprints*. The software therefore gets developed over a series of manageable chunks. Each sprint typically takes only a couple of weeks to complete. At the end of each sprint, stakeholders and team members meet to assess the progress made and the stakeholders suggest to the development team any changes needed to features that have already been developed and any overall improvements that they might feel necessary.

In the scrum model, the team members assume three fundamental roles—software owner, scrum master, and team member. The software owner is responsible for communicating the customers vision of the software to the development team. The scrum master acts as a liaison between the software owner and the team, thereby facilitating the development work.

2.5 SPIRAL MODEL

This model gets its name from the appearance of its diagrammatic representation that looks like a spiral with many loops (see Figure 2.10). The exact number of loops of the spiral is not fixed and can vary from project to project. The number of loops shown in Figure 2.10 is just an example. Each loop of the spiral is called a *phase* of the software process. The exact number of phases through which the product is developed can be varied by the project manager depending upon the project risks. A prominent feature of the spiral model is handling unforeseen risks that can show up much after the project has started. In this context, please recollect that the prototyping model can be used effectively only when the risks in a project can be identified upfront before the development work starts. As we shall discuss, this model achieves this by incorporating much more flexibility compared to SDLC

other models.

While the prototyping model does provide explicit support for risk handling, the risks are assumed to have been identified completely before the project start. This is required since the prototype is constructed only at the start of the project. In contrast, in the spiral model prototypes are built at the start of every phase. Each phase of the model is represented as a loop in its diagrammatic representation. Over each loop, one or more features of the product are elaborated and analysed and the risks at that point of time are identified and are resolved through prototyping. Based on this, the identified features are implemented.

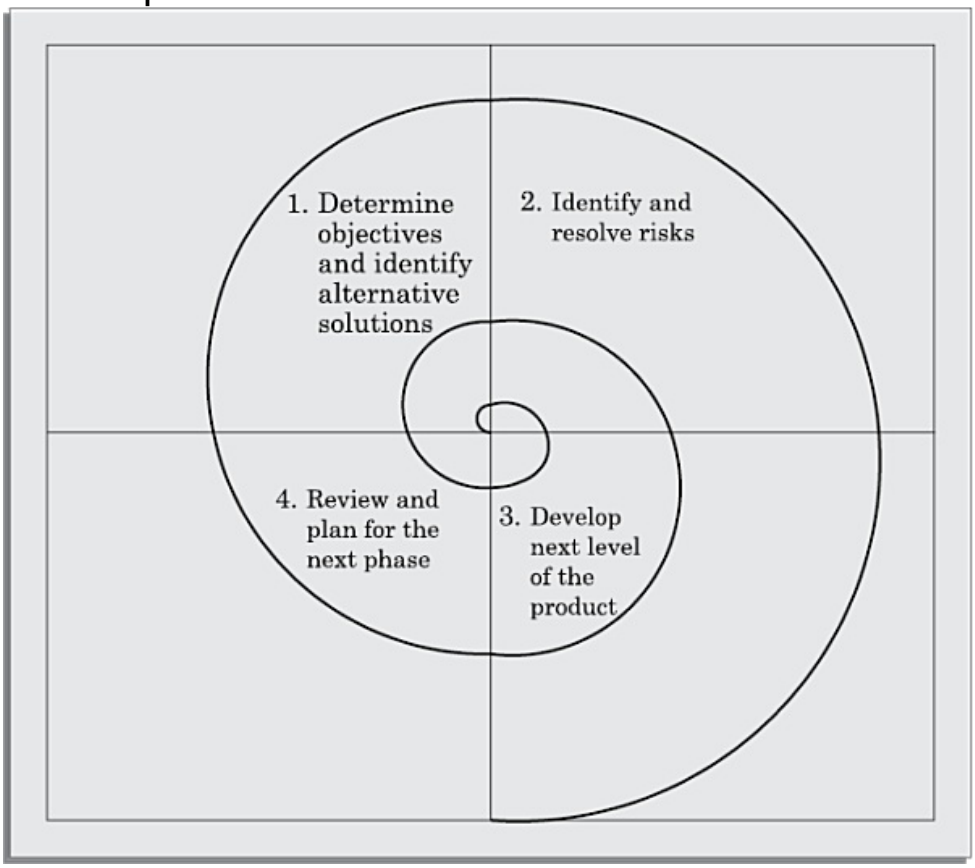

Figure 2.10: Spiral model of software development.

Risk handling in spiral model

A risk is essentially any adverse circumstance that might hamper the successful completion of a software project. As an example, consider a project for which a risk can be that data access from a remote database might be too slow to be acceptable by the customer. This risk can be resolved by building a prototype of the data access subsystem and experimenting with the exact access rate. If the data access rate is too slow, possibly a caching scheme can be implemented or a faster

communication scheme can be deployed to overcome the slow data access rate. Such risk resolutions are easier done by using a prototype as the pros and cons of an alternate solution scheme can evaluated faster and inexpensively, as compared to experimenting using the actual software application being developed. The spiral model supports coping up with risks by providing the scope to build a prototype at every phase of software development.

2.5.1 Phases of the Spiral Model

Each phase in this model is split into four sectors (or quadrants) as shown in Figure 2.10. In the first quadrant, a few features of the software are identified to be taken up for immediate development based on how crucial it is to the overall software development. With each iteration around the spiral (beginning at the center and moving outwards), progressively more complete versions of the software get built. In other words, implementation of the identified features forms a phase.

Quadrant 1: The objectives are investigated, elaborated, and analysed. Based on this, the risks involved in meeting the phase objectives are identified. In this quadrant, alternative solutions possible for the phase under consideration are proposed.

Quadrant 2: During the second quadrant, the alternative solutions are evaluated to select the best possible solution. To be able to do this, the solutions are evaluated by developing an appropriate prototype.

Quadrant 3: Activities during the third quadrant consist of developing and verifying the next level of the software. At the end of the third quadrant, the identified features have been implemented and the next version of the software is available.

Quadrant 4: Activities during the fourth quadrant concern reviewing the results of the stages traversed so far (i.e. the developed version of the software) with the customer and planning the next iteration of the spiral.

The radius of the spiral at any point represents the cost incurred in the project so far, and the angular dimension represents the progress made so far in the current phase.

In the spiral model of development, the project manager dynamically determines the number of phases as the project progresses. Therefore, in this model, the project manager plays the crucial role of tuning the model to

specific projects.

To make the model more efficient, the different features of the software that can be developed simultaneously through parallel cycles are identified. To keep our discussion simple, we shall not delve into parallel cycles in the spiral model.

Advantages/pros and disadvantages/cons of the spiral model

There are a few disadvantages of the spiral model that restrict its use to a only a few types of projects. To the developers of a project, the spiral model usually appears as a complex model to follow, since it is risk-driven and is more complicated phase structure than the other models we discussed. It would therefore be counterproductive to use, unless there are knowledgeable and experienced staff in the project. Also, it is not very suitable for use in the development of outsourced projects, since the software risks need to be continually assessed as it is developed.

In spite of the disadvantages of the spiral model that we pointed out, for certain categories of projects, the advantages of the spiral model can outweigh its disadvantages.

For projects having many unknown risks that might show up as the development proceeds, the spiral model would be the most appropriate development model to follow.

In this regard, it is much more powerful than the prototyping model. Prototyping model can meaningfully be used when all the risks associated with a project are known beforehand. All these risks are resolved by building a prototype before the actual software development starts.

Spiral model as a meta model

As compared to the previously discussed models, the spiral model can be viewed as a *meta model*, since it subsumes all the discussed models. For example, a single loop spiral actually represents the waterfall model. The spiral model uses the approach of the prototyping model by first building a prototype in each phase before the actual development starts. This prototypes are used as a risk reduction mechanism. The spiral model incorporates the systematic step- wise approach of the waterfall model. Also, the spiral model can be considered as supporting the evolutionary model—the

iterations along the spiral can be considered as evolutionary levels through which the complete system is built. This enables the developer to understand and resolve the risks at each evolutionary level (i.e. iteration along the spiral).

Case study 2.2

Galaxy Inc. undertook the development of a satellite-based communication between mobile handsets that can be anywhere on the earth. In contrast to the traditional cell phones, by using a satellite-based mobile phone a call can be established as long as both the source and destination phones are in the coverage areas of some base stations. The system would function through about six dozens of satellites orbiting the earth. The satellites would directly pick up the signals from a handset and beam signal to the destination handset. Since the foot prints of the revolving satellites would cover the entire earth, communication between any two points on the earth, even between remote places such as those in the Arctic ocean and Antarctica, would also be possible. However, the risks in the project are many, including determining how the calls among the satellites can be handed-off when they are themselves revolving at a very high speed. In the absence of any published material and availability of staff with experience in development of similar products, many of the risks cannot be identified at the start of the project and are likely to crop up as the project progresses. The software would require several million lines of code to be written. Galaxy Inc. decided to deploy the spiral model for software development after hiring highly qualified staff. To speed up the software development, independent parts of the software were developed through parallel cycles on the spiral. The cost and delivery schedule were refined many times, as the project progressed. The project was successfully completed after five years from start date

2.6 A COMPARISON OF DIFFERENT LIFE CYCLE MODELS

The classical waterfall model can be considered as the basic model and all other life cycle models as embellishments of this model. However, the classical waterfall model cannot be used in practical development projects, since this model supports no mechanism to correct the errors that are committed during any of the phases but detected at a later phase. This problem is overcome by the iterative waterfall model through the provision of feedback paths.

The iterative waterfall model is probably the most widely used software development model so far. This model is simple to understand and use. However, this model is suitable only for well-understood problems, and is not suitable for development of very large projects and projects that suffer from large number of risks.

The prototyping model is suitable for projects for which either the user requirements or the underlying technical aspects are not well understood, however all the risks can be identified before the project starts. This model is especially popular for development of the user interface part of projects.

The evolutionary approach is suitable for large problems which can be decomposed into a set of modules for incremental development and delivery. This model is also used widely for object-oriented development projects. Of course, this model can only be used if incremental delivery of the system is acceptable to the customer.

The spiral model is considered a *meta model* and encompasses all other life cycle models. Flexibility and risk handling are inherently built into this model. The spiral model is suitable for development of technically challenging and large software that are prone to several kinds of risks that are difficult to anticipate at the start of the project. However, this model is much more complex than the other models—this is probably a factor deterring its use in ordinary projects.

Let us now compare the prototyping model with the spiral model. The prototyping model can be used if the risks are few and can be determined at the start of the project. The spiral model, on the other hand, is useful when the risks are difficult to anticipate at the beginning of the project, but are likely to crop up as the development proceeds.

Let us compare the different life cycle models from the viewpoint of the customer. Initially, customer confidence is usually high on the development team irrespective of the development model followed. During the lengthy development process, customer confidence normally drops off, as no working software is yet visible. Developers answer customer queries using technical slang, and delays are announced. This gives rise to customer resentment. On the other hand, an evolutionary approach lets the customer experiment with a working software much earlier than the monolithic approaches. Another important advantage of the incremental model is that it reduces the customer's trauma of getting used to an entirely new system. The gradual introduction of the software via incremental phases provides time to the customer to adjust to the new software. Also, from the customer's financial view point, incremental development does not require a large upfront capital outlay. The customer can order the incremental versions as and when he can afford them.

2.6.1 Selecting an Appropriate Life Cycle Model for a Pro ject

We have discussed the advantages and disadvantages of the various life cycle models. However, how to select a suitable life cycle model for a specific project? The answer to this question would depend on several factors. A suitable life cycle model can possibly be selected based on an analysis of issues such as the following:

Characteristics of the software to be developed: The choice of the life cycle model to a large extent depends on the nature of the software that is being developed. For small services projects, the agile model is favoured. On the other hand, for product and embedded software development, the iterative waterfall model can be preferred. An evolutionary model is a suitable model for object-oriented development projects.

Characteristics of the development team: The skill-level of the team members is a significant factor in deciding about the life cycle model to use. If the development team is experienced in developing similar software, then even an embedded software can be developed using an iterative waterfall model. If the development team is entirely novice, then even a simple data processing application may require a prototyping model to be adopted.

Characteristics of the customer: If the customer is not quite familiar with computers, then the requirements are likely to change frequently as it would be difficult to form complete, consistent, and unambiguous requirements. Thus, a prototyping model may be necessary to reduce later change requests from the customers.

SUMMARY

- During the development of any type of software, adherence to a suitable process model has become universally accepted by software development organisations. Adoption of a suitable life cycle model is now accepted as a primary necessity for successful completion of projects.
- We discussed only the central ideas behind some important process models. Good software development organisations carefully and elaborately document the precise process model they follow and typically include the following in the document:

– Identification of the different phases.
– Identification of the different activities in each phase and the order in which they are carried out.

– The phase entry and exit criteria for different phases.
– The methodology followed to carry out the different activities.

- Adherence to a software life cycle model encourages the team members to perform various development activities in a systematic and disciplined manner. It also makes management of software development projects easier.
- The principle of detecting errors as close to their point of introduction as possible is known as phase containment of errors. Phase containment minimises the cost to fix errors.
- The classical waterfall model can be considered as the basic model and all other life cycle models are embellishments of this model. Iterative waterfall model has been the most widely used life cycle model so far, though the usage of RAD and agile models have been increasing.
- Different life cycle models have their own advantages and disadvantages. Therefore, an appropriate life cycle model should be chosen for the problem at hand. After choosing a basic life cycle model, software development organisations usually tailor the standard life cycle models according to their needs.
- Even though an organisation may follow whichever life cycle model is appropriate to a project, the final document should reflect as if the software was developed using the classical waterfall model. This makes it easier for the maintainers to understand the software documents.

EXERCISES

1. Choose the correct option for each of the following questions:
   (a) Which one of the following disadvantages may be experienced when a systematic development process model is adopted in preference over a build-and-fix style of development?
   (i) Increased documentation overhead
   (ii) Increased development cost
   (iii) Decreased maintainability
   (iv) Increased development time
   (b) A software process model represents which one of the following:
   (i) The way in which software is developed (ii) The way in which software processes data (iii) The way in which software is used
   (iv) The way in which software may fail

(c) In the waterfall SDLC model, unit testing is carried out during which one of the following phases?
(i) Coding (ii) Testing (iii) Design
(iv) Maintenance

(d) Which of the following activity spans all stages of a software development life cycle (SDLC)?
(i) Coding
(ii) Testing
(iii) Project management
(iv) Design

(e) The operation phase in the waterfall model is a synonym for which one of the following phases:
(i) Coding and unit testing phase
(ii) Integration and system testing phase
(iii) Maintenance phase
(iv) Design phase

(f) The implementation phase in the waterfall model is a synonym for which one of the following phases:
(i) Coding and unit testing phase
(ii) Integration and system testing phase
(iii) Maintenance phase
(iv) Design phase

(f) Unit testing is carried out in which phase of the waterfall model:
(i) Implementation phase
(ii) Testing phase
(iii) Maintenance phase
(iv) Design phase

(h) Which one of the following phases accounts for the the maximum effort during development of a typical software?
(i) Coding
(ii) Testing
(iii) Designing
(iv) Specification

(i) Which of the following is not a standard software development process model?
(i) Waterfall Model
(ii) Watershed Model
(iii) RAD Model

(iv) V-Model

(j) Which one of the following feedback paths is not present in an iterative waterfall model?
(i) Design phase to feasibility study phase
(ii) Implementation phase to design phase
(iii) Implementation phase to requirements specification phase
(iv) Design phase to requirements specification phase

(k) Which one of the following is a suitable SDLC model for developing a moderate- sized software for which the customer is not clear about his exact requirements?
(i) RAD model
(ii) V-model
(iii) Iterative waterfall model
(iv) Classical waterfall model

(l) Which one of the following SDLC models would be suitable for use in a project involving customisation of a computer communication package? Assume that the project would be manned by experience personnel. The schedule for the project has been very aggressively set?
(i) Spiral model
(ii) Iterative waterfall model
(iii) RAD model
(iv) Agile model

(m) Which one of the following life cycle models lacks the characteristics of iterative software development?
(i) Spiral model
(ii) Prototyping model
(iii) Classical waterfall model
(iv) Evolutionary model

(n) Which one of the following life cycle models does not involve constructing a prototype any time during software development?
(i) Spiral model
(ii) Prototyping model
(iii) RAD model
(iv) Evolutionary model

(o) GUI part of an application software is typically developed using which life cycle model?
(i) Iterative waterfall model
(ii) Spiral model

(iii) Prototyping model
(iv) Evolutionary model

(p) Which of the following is not a characteristic of the agile model of software development?
(i) Prototype construction
(ii) Evolutionary development
(iii) Iterative development
(iv) Periodic delivery of working software

(q) Which one of the following SDLC models can be considered to be more effective for determination of the exact customer requirements?
(i) Iterative waterfall model
(ii) V-model
(iii) Prototyping model
(iv) Classical waterfall model

(r) Change requests from customers later in the development cycle are easiest to handle in which of the following life cycle models?
(i) Iterative waterfall model
(ii) Prototyping model
(iii) V-model
(iv) Evolutionary model

(s) Assume that you are the project manager of a development project for a data processing application in which the user requirements for the GUI part are not very clear. Which life cycle model would you use to develop the GUI part?
(i) Classical waterfall model
(ii) Iterative waterfall model
(iii) Prototyping model
(iv) Spiral model

(t) The angular dimension of the spiral model does not represent which one of the followings?
(i) Cost incurred so far
(ii) Number of features implemented so far
(iii) Progress in the implementation of the current feature
(iv) Number of risks that have been resolved so far

(u) The radial dimension of the spiral model represents which one of the followings?
(i) Cost incurred so far
(ii) Number of features implemented so far

(iii) Progress in the implementation of the current feature
(iv) Number of risks that have been resolved so far

2. What do you understand by the term software life cycle? Why is it necessary to model software life cycle and to document it?
3. What do you understand by the term software development life cycle model (SDLC)?
   What problems might a software development organisation face if it is not following any SDLC for development of a large-sized software?
4. What problems would a software development organisation face if it does not have a documented process model, and therefore follows only an informal one?
5. Are the terms SDLC and software development process synonymous? Explain your answer.
6. Why is it important for an organisation to properly document its development process?
7. (a) Mention the major activities that are undertaken during the development of a software software.
   (b) Name an activity that spans all the development phases.
8. What do you mean by a software development process? What is the difference between a methodology and a process? Explain your answer using a suitable example.
9. Which are the major phases in the waterfall model of software development? Which phase consumes the maximum effort for developing a typical software?
10. Why is the classical waterfall model called an idealistic development model? Does this model of development has any practical use at all?
11. Consider the following assertion: "The classical waterfall model is an idealistic model".
    Based on this assertion, answer the following:
    (a) Justify why the above assertion is true.
    (b) Even if the classical waterfall model is an idealistic model, is there any practical use of this model at all? Explain your answer.
12. What is the difference between programming-in-the-small and programming-in- the-large? Is using waterfall SDLC model a good idea for programming-in-the-small? Explain your answer.
13. Draw a schematic diagram to represent the iterative waterfall model of software development. On your diagram represent the following:
    (a) The phase entry and exit criteria for each phase.

(b) The deliverables that need to be produced at the end of each phase.

14. What are the objectives of the feasibility study phase of software development? Explain the important activities that are carried out during the feasibility study phase of a software development project. Who carries out these activities? Mention suitable phase entry and exit criteria for this phase.
15. Give an example of a software development project for which the iterative waterfall model is not suitable. Briefly justify your answer.
16. In practical software development projects using iterative waterfall SDLC, why do different phases overlap? Explain the effort distribution over different phases.
17. Identify five reasons as to why the customer requirements may change after the requirements phase is complete and the SRS document has been signed off.
18. Identify the criteria based on which a suitable life cycle model can be chosen for a given project development. Illustrate your answer using suitable examples.
19. Briefly explain the important differences and similarities between the incremental and evolutionary models of SDLCs.
20. What do you understand by "build-and-fix" style of software development? Diagra- mmatically depict the typical activities in this style of development and their ordering. Identify at least four major problems that would arise, if a large professional software development project is undertaken using a "build-and-fix" style of software development.
21. State whether the following statements are **TRUE** o r **FALSE**. Give reasons behind your answers.
    (a) If the phase containment of errors principle is not followed during software development, then development cost would increase.
    (b) Evolutionary life cycle model would be appropriate to develop a software that appears to be beset with a large number of risks.
    (c) The number of phases in the spiral life cycle model is not fixed and is normally determined by the project managers as the project progresses.
    (d) The primary purpose of phase containment of errors is to develop an error-free software.
    (e) Development of a software using the prototyping life cycle model is always more expensive than development of the same software using the iterative waterfall model due to the additional cost incurred to

construct a throw-away prototype.

(f) When a large software is developed by a commercial software development house using the iterative waterfall model, there do not exist precise points of time at which transitions from one phase to another take place.

(g) Among all phases of software development, an undetected error from the design phase that ultimately gets detected during the system acceptance test costs the maximum.

(h) If a team developing a moderate sized software product does not care about phase containment of errors, it can still produce a reliable software, *al beit* at a higher cost compared to the case where it attempts phase containment of errors.

(i) The angular dimension in a spiral model of software development indicates the total cost incurred in the project till that time.

(j) When the spiral model is used in a software development project, the number of loops in the spiral is fixed by the project manager during the project planning stage.

(k) RAD would be a suitable life cycle model for developing a commercial operating system.

(l) RAD is a suitable process model to use for developing a safety-critical application such as a controller for a nuclear reactor.

22. What do you understand by the "99 per cent complete" syndrome that software project managers sometimes face? What are its underlying causes? What problems does it create for project management? What are its remedies?

23. While using the iterative waterfall model to develop a commercial software for an industrial application, discuss how the effort spent on the different phases is spread over time.

24. Which life cycle model would you follow for developing software for each of the following applications? Mention the reasons behind your choice of a particular life cycle model.

(a) A well-understood data processing application.

(b) A new software that would connect computers through satellite communication.

Assume that your team has no previous experience in developing satellite communication software.

(c) A software that would function as the controller of a telephone switching system.

(d) A new library automation software that would link various libraries in the city.
(e) An extremely large software that would provide, monitor, and control cellular communication among its subscribers using a set of revolving satellites.
(f) A new text editor.
(g) A compiler for a new language.
(h) An object-oriented software development effort.
(i) The graphical user interface part of a large software.

25. Briefly explain the V SDLC model and answer the following specific questions pertaining to the V SDLC.
(a) What are the strengths and weaknesses of the V-model?
(b) Outline the similarities and differences of the V-model with the iterative waterfall model.
(c) Give an example of a development project for which V-model can be considered appropriate and also give an example of a project for which it would be clearly inappropriate.
26. Briefly explain the V SDLC model. Identify why for developing safety-critical software, the V SDLC model is usually considered suitable.
27. With respect to the prototyping model for software development, answer the following: (a) What is a prototype?
(b) Is it necessary to develop a prototype for all types of projects?
(c) If you answer to part (b) of the question is no, then mention under what circumstances is it beneficial to construct a prototype.
(d) If your answer to part (b) of the question is yes, then explain does construction of a prototype always increase the overall cost of software development.
28. If the prototyping model is being used in a development project of moderate size, is it necessary to develop an SRS document? Justify your answer.
29. Consider that a software development project that is beset with many risks. But, assume that it is not possible to anticipate all the risks in the project at the start of the project and some of the risks can only be identified much after the development is underway. As the project manager recommend the use of the prototyping or the spiral model? Explain your answer.
30. What are the major advantages of first constructing a working prototype before starting to develop the actual software? What are the

disadvantages of this approach?

31. Explain how a software development effort is initiated and finally terminated in the spiral model.
32. Suppose a travel agency needs a software for automating its book-keeping activities. The set of activities to be automated are rather simple and are at present being carried out manually. The travel agency has indicated that it is unsure about the type of user interface which would be suitable for its employees and its customers. Would it be proper for a development team to use the spiral model for developing this software?
33. Explain why the spiral life cycle model is considered to be a meta model.
34. Both the prototyping model as well as the spiral model have been designed to handle risks. Identify how exactly risk is handled in each. How do these two models can be compared with respect to their risk handling capabilities?
35. Explain with suitable examples, the types of software development for which the spiral model is suitable. Is the number of loops of the spiral fixed for different development projects? If not, explain how the number of loops in the spiral is determined.
36. Discuss the relative merits of developing a throw-away prototype to resolve risks versus refining a developed prototype into the final software.
37. Answer the following questions, using one sentence for each:
    (a) How are the risks associated with a project handled in the spiral model of software development?
    (b) Which types of risks are be better handled using the spiral model compared to the prototyping model?
    (c) Give an example of a project where the spiral model can be meaningfully be deployed.
38. Compare the relative advantages of using the iterative waterfall model and the spiral model of software development for developing an MIS application. Explain with the help of one suitable example each, the type of project for which you would use the waterfall model of software development, and the type of project for which you would use the spiral model.
39. Briefly discuss the evolutionary process model. Explain using suitable examples the types of software development projects for which the evolutionary life cycle model is suitable. Compare the advantages and disadvantages of this model with the iterative waterfall model.

40. Assume that a software development company is already experienced in developing payroll software and has developed similar software for several customers (organisations). Assume that the software development company has received a request from a certain customer (organisation), which was still using manually processing of its pay rolls. For developing a payroll software for this organisation, which life cycle model should be used? Justify your answer.
41. Explain why it may not be prudent to use the spiral model in the development of any large software.
42. Instead of having a one time testing of a software at the end of its development, why are three different levels of testing—unit testing, integration testing, and system testing—are necessary? What is the main purpose of each of these different levels of testing?
43. What do you understand by the term phase containment of errors? Why is phase containment of errors is considered to be important? How can phase containment of errors be achieved in a software development project?
44. Irrespective of whichever life cycle model is followed for developing a software, why is it necessary for the final documents to describe the software as if it were developed using the classical waterfall model?
45. What are the major shortcomings of the iterative waterfall model? Name the life cycle models that overcome any of the specific shortcomings. How are the shortcomings overcome in those models?
46. For which types of development projects is the V-model appropriate? Briefly explain the V-model and point out its strengths and weaknesses.
47. Identify the main motivation and goals behind the development of the RAD model. How does the model help achieve the identified goals?
48. Explain the following aspects of rapid application development (RAD) SDLC model:
    ](a) What is a time box in a RAD model?
    (b) How does RAD facilitate faster development?
    (c) Identify the key differences between the RAD model and the prototyping model.
    (d) Identify the types of projects for which RAD model is suitable and the types for which it is not suitable.
49. Suggest a suitable life-cycle model for a software project which your organisation has undertaken on behalf of certain customer who is unsure of his requirements and is likely to change his requirements frequently,

since the business process of the customer (organisation) is of late changing rapidly. Give the reasonings behind your answer.

50. Draw a labelled schematic diagram to represent the spiral model of software development.

Is the number of loops of the spiral fixed? If your answer is affirmative, write down the number of the loops that the spiral has. If your answer is negative, explain how and on what basis the number of loops of the spiral can be determined.

51. Assume that you are the project manager of a development team that is using the iterative waterfall model for developing a certain software. Would you recommend that the development team should start a development phase only after the previous phase is fully complete? Explain your answer.

52. Identify the major differences between the iterative and evolutionary SDLCs.

53. Explain how the characteristics of the product, the development team, and the customer influence the selection of an appropriate SDLC for a project.

54. With respect to the rapid application development (RAD) model, answer the following:

(a) Explain the different life cycle activities that are carried out in the RAD model.

(b) How does RAD model help accommodate change requests late in the development.

(c) How does RAD help in faster software development.

(d) Give examples of two projects for which RAD would be a suitable model for development.

(e) Point out the advantages and disadvantages of the RAD model as compared to (i) prototyping model and (ii) evolutionary model.

(f) Point a disadvantage of the RAD model as compared to iterative waterfall model.

(g) Identify the characteristics that make a project suited to RAD style of development.

(h) Identify the characteristics that make a project unsuited to RAD style of development.

55. Identified the important factors that influence the choice of a suitable SDLC model for a software development project.

56. Explain the important features of the agile software development

model.

(a) Compare the advantages and disadvantages of the agile model with iterative waterfall and the exploratory programming model.

(b) Is the agile life cycle model suitable for development of embedded software? Briefly justify your answer.

57. Briefly explain the agile software development model. Give an example of a project for which the agile model would be suitable and one project project for which the agile model would not be appropriate.
58. Explain the similarities in the objectives and practices of the RAD, agile, and extreme programming (XP) models of software development. Also explain the dissimilarities among these three models.
59. Briefly discuss the RAD model. Identify the main advantages of RAD model as compared to the iterative waterfall model. How does RAD model achieve faster development as compared to iterative waterfall model?
60. Compare the relative advantages of RAD, iterative waterfall, and the evolutionary models of software development.
61. Identify and explain the important best practices that have been incorporated in the extreme programming model.
62. Using one or two sentences explain the important shortcomings of the classical waterfall model that RAD, agile, and extreme programming (XP) models of software development address.
63. Briefly explain the extreme programming (XP) SDLC model. Identify the key principles that need to be practised to the extreme in XP. What is a spike in XP? Why is it required?
64. Identify how the agile SDLCs achieve reductions to the development time and cost. Are there any pitfalls of achieving cost and time reductions this way?
65. Suppose a development project has been undertaken by a company for customising one of its existing software on behalf of a specific customer. Identify two major advantages of using an agile model over the iterative waterfall model.
66. Which life cycle model would you recommend for developing an object-oriented software?
Justify your answer.
67. What do you understand by pairwise programming? What are its advantages over traditional programming?
68. Graphically represent the activities that are undertaken in a typical

build and fix style of software development and show the ordering among the activities.

69. Analyse and graphically represent the life cycle model of an open source software such as Linux or Apache.

1 Dictionary meaning of *agile*: To move quickly

Chapter

3

SOFTWARE PROJECT MANAGEMENT

Effective project management is crucial to the success of any software development project. In the past, several projects have failed not for want of competent technical professionals neither for lack of resources, but due to the use of faulty project management practices. Therefore, it is important to carefully learn the latest software project management techniques.

Software project management is a very vast topic. In fact, a full semester teaching can be conducted on effective techniques for software project management. However, in this chapter, we shall restrict ourselves to only some basic issues. Let us first understand what exactly is the principal goal of software project management.

The main goal of software project management is to enable a group of developers to work effectively towards the successful completion of a project.

As can be inferred from the above definition, project management involves use of a set of techniques and skills to steer a project to success. Before focusing on these project management techniques, let us first figure out who should be responsible for managing a project. A project manager is usually an experienced member of the team who essentially works as the administrative leader of the team. For small software development projects, a single member of the team assumes the responsibilities for both project management and technical management. For large projects, a different member of the team (other than the project manager) assumes the responsibility of technical leadership. The responsibilities of the technical leader includes addressing issues such as which tools and techniques to use in the project, high-level solution to the problem, specific algorithms to use, etc.

In this chapter, we first investigate why management of software projects is much more complex than managing many other types of projects. Subsequently, we outline the main responsibilities and activities of a software project manager. Next, we provide an overview of the project planning activity. We then discuss estimation and scheduling techniques. Finally, we provide an overview of the risk and configuration management activities.

3.1 SOFTWARE PROJECT MANAGEMENT COMPLEXITIES

Management of software projects is much more complex than management of many other types of projects. The main factors contributing to the complexity of managing a software project, as identified by [Brooks75], are the following:

Invisibility: Software remains invisible, until its development is complete and it is operational. Anything that is invisible, is difficult to manage and control. Consider a house building project. For this project, the project manger can very easily assess the progress of the project through a visual examination of the building under construction. Therefore, the manager can closely monitor the progress of the project, and take remedial actions whenever he finds that the progress is not as per plan. In contrast, it becomes very difficult for the manager of a software project to assess the progress of the project due to the invisibility of software. The best that he can do perhaps is to monitor the milestones that have been completed by the development team and the documents that have been produced—which are rough indicators of the progress achieved.

Invisibility of software makes it difficult to assess the progress of a project and is a major cause for the complexity of managing a software project.

Changeability: Because the software part of any system is easier to change as compared to the hardware part, the software part is the one that gets most frequently changed. This is especially true in the later stages of a project. As far as hardware development is concerned, any late changes to the specification of the hardware system under development usually amounts to redoing the entire project. This makes late changes to a hardware project prohibitively expensive to carry out. This possibly is a reason why requirement changes are frequent in software projects. These changes usually arise from changes to the business practices, changes to the hardware or underlying software (e.g. operating system, other applications), or just because the client changes his mind.

Frequent changes to the requirements and the invisibility of software are possibly the two major factors making software project management a complex task.

Complexity: Even a moderate sized software has millions of parts (functions) that interact with each other in many ways—data coupling, serial and concurrent runs, state transitions, control dependency, file sharing, etc. Due to the inherent complexity of the functioning of a software product in terms of the basic parts making up the software, many types of risks are associated with its development. This makes managing these projects much more difficult as compared to many other kinds of projects.

Uniqueness: Every software project is usually associated with many unique features or situations. This makes every project much different from the others. This is unlike projects in other domains, such as car manufacturing or steel manufacturing where the projects are more predictable. Due to the uniqueness of the software projects, a project manager in the course of a project faces many issues that are quite unlike the others he had encountered in the past. As a result, a software project manager has to confront many unanticipated issues in almost every project that he manages.

Exactness of the solution: Mechanical components such as nuts and bolts typically work satisfactorily as long as they are within a tolerance of 1 per cent or so of their specified sizes. However, the parameters of a function call in a program are required to be in complete conformity with the function definition. This requirement not only makes it difficult to get a software product up and working, but also makes reusing parts of one software product in another difficult. This requirement of exact conformity of the parameters of a function introduces additional risks and contributes to the complexity of managing software projects.

Team-oriented and intellect-intensive work: Software development projects are akin to research projects in the sense that they both involve team-oriented, intellect-intensive work. In contrast, projects in many domains are labour-intensive and each member works in a high degree of autonomy. Examples of such projects are planting rice, laying roads, assembly-line manufacturing, constructing a multi-storeyed building, etc. In a software development project, the life cycle activities not only highly intellect-intensive, but each member has to typically interact, review, and interface with several other members, constituting another dimension of complexity of software projects.

3 . 2 RESPONSIBILITIES OF A SOFTWARE PROJECT MANAGER

In this section, we examine the principal job responsibilities of a project manager and the skills necessary to accomplish those.

3.2.1 Job Responsibilities for Managing Software Projects

A software project manager takes the overall responsibility of steering a project to success. This surely is a very hazy job description. In fact, it is very difficult to objectively describe the precise job responsibilities of a project manager. The job responsibilities of a project manager ranges from invisible activities like building up of team morale to highly visible customer presentations. Most managers take the responsibilities for project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, managerial report writing and presentation, and interfacing with clients. These activities are certainly numerous and varied. We can still broadly classify these activities into two major types—project planning and project monitoring and control.

We can broadly classify a project manager's varied responsibilities into the following two major categories:
- Project planning, and
- Project monitoring and control.

In the following subsections, we give an overview of these two classes of responsibilities. Later on, we shall discuss them in more detail.

Project planning: Project planning is undertaken immediately after the feasibility study phase and before the starting of the requirements analysis and specification phase.

Project planning involves estimating several characteristics of a project and then planning the project activities based on these estimates made.

The initial project plans are revised from time to time as the project progresses and more project data become available.

Project monitoring and control: Project monitoring and control activities are undertaken once the development activities start.

The focus of project monitoring and control activities is to ensure that the software development proceeds as per plan.

While carrying out project monitoring and control activities, a project manager may sometimes find it necessary to change the plan to cope up with specific situations at hand.

3.2.2 Skills Necessary for Managing Software Projects

A theoretical knowledge of various project management techniques is certainly important to become a successful project manager. However, a purely theoretical knowledge of various project management techniques would hardly make one a successful project manager. Effective software project management calls for good qualitative judgment and decision taking capabilities. In addition to having a good grasp of the latest software project management techniques such as cost estimation, risk management, and configuration management, etc., project managers need good communication skills and the ability to get work done. Some skills such as tracking and controlling the progress of the project, customer interaction, managerial presentations, and team building are largely acquired through experience. Never the less, the importance of a sound knowledge of the prevalent project management techniques cannot be overemphasized. The objective of the rest of this chapter is to introduce the reader to the same.

Three skills that are most critical to successful project management are the following:
- Knowledge of project management techniques.
- Decision taking capabilities.
- Previous experience in managing similar projects.

With this brief discussion on the overall responsibilities of a software project manager and the skills necessary to accomplish these, in the next section we examine some important issues in project planning.

3.3 PROJECT PLANNING

Once a project has been found to be feasible, software project managers undertake project planning.

Project planning is undertaken and completed before any development activity starts.

Project planning requires utmost care and attention since commitment to unrealistic time and resource estimates result in schedule slippage. Schedule delays can cause customer dissatisfaction and adversely affect team morale. It can even cause project failure. For this reason, project planning is undertaken by the project managers with utmost care and attention.

However, for effective project planning, in addition to a thorough knowledge of the various estimation techniques, past experience is crucial.

During project planning, the project manager performs the following activities. Note that we have given only a very brief description of the activities. We discuss these in the later section in more detail.

Estimation: The following project attributes are estimated.

- **Cost:** How much is it going to cost to develop the software product?
- **Duration:** How long is it going to take to develop the product?
- **Effort:** How much effort would be necessary to develop the product?

The effectiveness of all later planning activities such as scheduling and staffing are dependent on the accuracy with which these three estimations have been made.

Scheduling: After all the necessary project parameters have been estimated, the schedules for manpower and other resources are developed.

Staffing: Staff organisation and staffing plans are made.

Risk management : This includes risk identification, analysis, and abatement planning.

Miscellaneous plans: This includes making several other plans such as quality assurance plan, and configuration management plan, etc.

Figure 3.1 shows the order in which the planning activities are undertaken. Observe that size estimation is the first activity that a project manager undertakes during project planning.

Size is the most fundamental parameter based on which all other estimations and project plans are made.

As can be seen from Figure 3.1, based on the size estimation, the effort required to complete a project and the duration over which the development is to be carried out are estimated. Based on the effort estimation, the cost of the project is computed. The estimated cost forms the basis on which price negotiations with the customer is carried out. Other planning activities such as staffing, scheduling etc. are undertaken based on the effort and duration estimates made. In Section 3.7, we shall discuss a popular technique for estimating the project parameters. Subsequently, we shall discuss the staffing and scheduling issues.

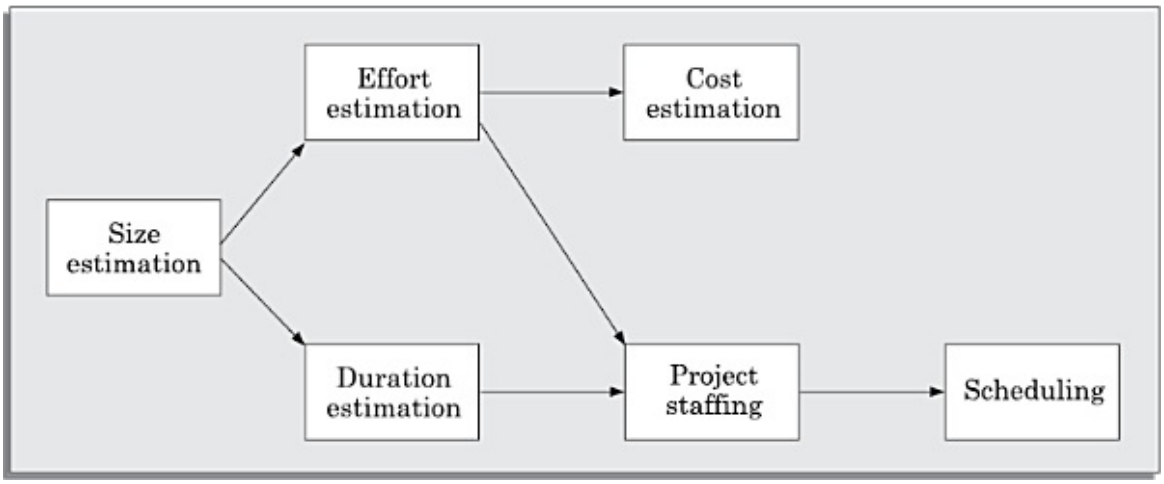

Figure 3.1: Precedence ordering among planning activities.

3.3.1 Sliding Window Planning

It is usually very difficult to make accurate plans for large projects at project initiation. A part of the difficulty arises from the fact that large projects may take several years to complete. As a result, during the span of the project, the project parameters, scope of the project, project staff, etc., often change drastically resulting in the initial plans going haywire. In order to overcome this problem, sometimes project managers undertake project planning over several stages. That is, after the initial project plans have been made, these are revised at frequent intervals. Planning a project over a number of stages protects managers from making big commitments at the start of the project. This technique of staggered planning is known as sliding window planning.

> In the sliding window planning technique, starting with an initial plan, the project is planned more accurately over a number of stages.

At the start of a project, the project manager has incomplete knowledge about the nitty-gritty of the project. His information base gradually improves as the project progresses through different development phases. The complexities of different project activities become clear, some of the anticipated risks get resolved, and new risks appear. The project parameters are re-estimated periodically as understanding grows and also aperiodically as project parameters change. By taking these developments into account, the project manager can plan the subsequent activities more accurately and with increasing levels of confidence.

3.3.2 The SPMP Document of Project Planning

Once project planning is complete, project managers document their plans in a software project management plan (SPMP) document. Listed below are the different items that the SPMP document should discuss. This list can be used as a possible organisation of the SPMP document.

Organisation of the software project management plan (SPMP) document

1. **Introduction**
   (a) Objectives
   (b) Major Functions
   (c) Performance Issues
   (d) Management and Technical Constraints
2. **Project estimates**
   (a) Historical Data Used
   (b) Estimation Techniques Used
   (c) Effort, Resource, Cost, and Project Duration Estimates
3. **Schedule**
   (a) Work Breakdown Structure
   (b) Task Network Representation
   (c) Gantt Chart Representation
   (d) PERT Chart Representation
4. **Project resources**
   (a) People
   (b) Hardware and Software
   (c) Special Resources
5. **Staff organisation**
   (a) Team Structure
   (b) Management Reporting
6. **Risk management plan**
   (a) Risk Analysis
   (b) Risk Identification
   (c) Risk Estimation
   (d) Risk Abatement Procedures
7. **Project tracking and control plan**
   (a) Metrics to be tracked
   (b) Tracking plan
   (c) Control plan
8. **Miscellaneous plans**

(a) Process Tailoring
(b) Quality Assurance Plan
(c) Configuration Management Plan
(d) Validation and Verification
(e) System Testing Plan
(f) Delivery, Installation, and Maintenance Plan

3.4 METRICS FOR PROJECT SIZE ESTIMATION

As already mentioned, accurate estimation of project size is central to satisfactory estimation of all other project parameters such as effort, completion time, and total project cost. Before discussing the available metrics to estimate the size of a project, let us examine what does the term "project size" exactly mean. The size of a project is obviously not the number of bytes that the source code occupies, neither is it the size of the executable code.

The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.

Currently, two metrics are popularly being used to measure size—lines of code (LOC) and function point (FP). Each of these metrics has its own advantages and disadvantages. These are discussed in the following subsection. Based on their relative advantages, one metric may be more appropriate than the other in a particular situation.

3.4.1 Lines of Code (LOC)

LOC is possibly the simplest among all metrics available to measure project size. Consequently, this metric is extremely popular. This metric measures the size of a project by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, comment lines, and header lines are ignored.

Determining the LOC count at the end of a project is very simple. However, accurate estimation of LOC count at the beginning of a project is a very difficult task. One can possibly estimate the LOC count at the starting of a project, only by using some form of systematic guess work. Systematic guessing typically involves the following. The project manager divides the problem into modules, and each module into sub-modules and so on, until the LOC of the leaf-level modules are small enough to be predicted. To be able to predict the LOC count for the various leaf-level modules sufficiently

accurately, past experience in developing similar modules is very helpful. By adding the estimates for all leaf level modules together, project managers arrive at the total size estimation. In spite of its conceptual simplicity, LOC metric has several shortcomings when used to measure problem size. We discuss the important shortcomings of the LOC metric in the following subsections:

LOC is a measure of coding activity alone. A good problem size measure should consider the total effort needed to carry out various life cycle activities (i.e. specification, design, code, test, etc.) and not just the coding effort. LOC, however, focuses on the coding activity alone—it merely computes the number of source lines in the final program. We have already discussed in Chapter 2 that coding is only a small part of the overall software development effort.

> The implicit assumption made by the LOC metric is that the overall product development effort is solely determined from the coding effort alone is flawed.

The presumption that the total effort needed to develop a project is proportional to the coding effort is easily countered by noting the fact that even when the design or testing issues are very complex, the code size might be small and vice versa. Thus, the design and testing efforts can be grossly disproportional to the coding effort. Code size, therefore, is obviously an improper indicator of the problem size.

LOC count depends on the choice of specific instructions: LOC gives a numerical value of problem size that can vary widely with coding styles of individual programmers. By coding style, we mean the choice of code layout, the choice of the instructions in writing the program, and the specific algorithms used. Different programmers may lay out their code in very different ways. For example, one programmer might write several source instructions on a single line, whereas another might split a single instruction across several lines. Unless this issue is handled satisfactorily, there is a possibility of arriving at very different size measures for essentially identical programs. This problem can, to a large extent, be overcome by counting the language tokens in a program rather than the lines of code. However, a more intricate problem arises due to the specific choices of instructions made in writing the program. For example, one programmer may use a switch statement in writing a C program and another may use a sequence of if ... then ... else ... statements. Therefore, the following can easily be concluded.

> Even for the same programming problem, different programmers might come up with

programs having very different LOC counts. This situation does not improve, even if language tokens are counted instead of lines of code.

LOC measure correlates poorly with the quality and efficiency of the code: Larger code size does not necessarily imply better quality of code or higher efficiency. Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set or use improper algorithms. In fact, it is true that a piece of poor and sloppily written piece of code can have larger number of source instructions than a piece that is efficient and has been thoughtfully written. Calculating productivity as LOC generated per man-month may encourage programmers to write lots of poor quality code rather than fewer lines of high quality code achieve the same functionality.

LOC metric penalises use of higher-level programming languages and code reuse: A paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower. This would show up as smaller program size, and in turn, would indicate lower effort! Thus, if managers use the LOC count to measure the effort put in by different developers (that is, their productivity), they would be discouraging code reuse by developers. Modern programming methods such as object-oriented programming and reuse of components makes the relationships between LOC and other project attributes even less precise.

LOC metric measures the lexical complexity of a program and does not address the more important issues of logical and structural complexities: Between two programs with equal LOC counts, a program incorporating complex logic would require much more effort to develop than a program with very simple logic. To realise why this is so, imagine the effort that would be required to develop a program having multiple nested loops and decision constructs and compare that with another program having only sequential control flow.

It is very difficult to accurately estimate LOC of the final program from problem specification: As already discussed, at the project initiation time, it is a very difficult task to accurately estimate the number of lines of code (LOC) that the program would have after development. The LOC count can accurately be computed only after the code has fully been developed. Since project planning is carried out even before any development activity starts, the LOC metric is of little use to the project managers during project planning.

From the project managers perspective, the biggest shortcoming of the LOC metric is that the LOC count is very difficult to estimate during project planning stage, and can only be accurately computed after the software development is complete.

3.4.2 Function Point (FP) Metric

Function point metric was proposed by Albrecht in 1983. This metric overcomes many of the shortcomings of the LOC metric. Since its inception in late 1970s, function point metric has steadily gained popularity. Function point metric has several advantages over LOC metric. One of the important advantages of the function point metric over the LOC metric is that it can easily be computed from the problem specification itself. Using the LOC metric, on the other hand, the size can accurately be determined only after the product has fully been developed.

The conceptual idea behind the function point metric is the following. The size of a software product is directly dependent on the number of different high-level functions or features it supports. This assumption is reasonable, since each feature would take additional effort to implement.

Conceptually, the function point metric is based on the idea that a software product supporting many features would certainly be of larger size than a product with less number of features.

Though each feature takes some effort to develop, different features may take very different amounts of efforts to develop. For example, in a banking software, a function to display a help message may be much easier to develop compared to say the function that carries out the actual banking transactions. This has been considered by the function point metric by counting the number of input and output data items and the number of files accessed by the function. The implicit assumption made is that the more the number of data items that a function reads from the user and outputs and the more the number of files accessed, the higher is the complexity of the function. Now let us analyse why this assumption must be intuitively correct. Each feature when invoked typically reads some input data and then transforms those to the required output data. For example, the query book feature (see Figure 3.2) of a Library Automation Software takes the name of the book as input and displays its location in the library and the total number of copies available. Similarly, the issue book and the return book features produce their output based on the corresponding input data. It can therefore be argued that the computation of the number of input and output data items

would give a more accurate indication of the code size compared to simply counting the number of high-level functions supported by the system.

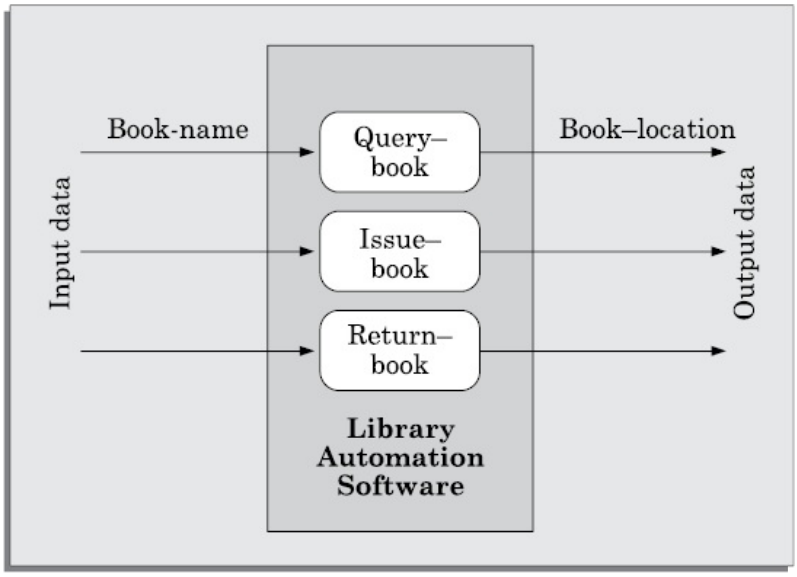

Figure 3.2: System function as a mapping of input data to output data.

Albrecht postulated that in addition to the number of basic functions that a software performs, size also depends on the number of files and the number of interfaces that are associated with the software. Here, interfaces refer to the different mechanisms for data transfer with external systems including the interfaces with the user, interfaces with external computers, etc.

Function point (FP) metric computation

The size of a software product (in units of function points or FPs) is computed using different characteristics of the product identified in its requirements specification. It is computed using the following three steps:

- **Step 1:** Compute the unadjusted function point (UFP) using a heuristic expression.
- **Step 2:** Refine UFP to reflect the actual complexities of the different parameters used in UFP computation.
- **Step 3:** Compute FP by further refining UFP to account for the specific characteristics of the project that can influence the entire development effort.

We discuss these three steps in more detail in the following.

Step 1: UFP computation

The unadjusted function points (UFP) is computed as the weighted sum of five characteristics of a product as shown in the following expression. The weights associated with the five characteristics were determined empirically by Albrecht through data gathered from many projects.

```
UFP = (Number of inputs)*4 + (Number of outputs)*5 +
 (Number of inquiries)*4 + (Number of files)*10 +
 (Number of interfaces)*10                    (3.1)
```

The meanings of the different parameters of Eq. 3.1 are as follows:

1. **Number of inputs:** Each data item input by the user is counted. However, it should be noted that data inputs are considered different from user inquiries. Inquiries are user commands such as print-account-balance that require no data values to be input by the user. Inquiries are counted separately (see the third point below). It needs to be further noted that individual data items input by the user are not simply added up to compute the number of inputs, but related inputs are grouped and considered as a single input. For example, while entering the data concerning an employee to an employee pay roll software; the data items name, age, sex, address, phone number, etc. are together considered as a single input. All these data items can be considered to be related, since they describe a single employee.
2. **Number of outputs:** The outputs considered include reports printed, screen outputs, error messages produced, etc. While computing the number of outputs, the individual data items within a report are not considered; but a set of related data items is counted as just a single output.
3. **Number of inquiries:** An inquiry is a user command (without any data input) and only requires some actions to be performed by the system. Thus, the total number of inquiries is essentially the number of distinct interactive queries (without data input) which can be made by the users. Examples of such inquiries are print account balance, print all student grades, display rank holders' names, etc.
4. **Number of files:** The files referred to here are logical files. A logical file represents a group of logically related data. Logical files include data structures as well as physical files.
5. **Number of interfaces:** Here the interfaces denote the different mechanisms that are used to exchange information with other

systems. Examples of such interfaces are data files on tapes, disks, communication links with other systems, etc.

Step 2: Refine parameters

UFP computed at the end of step 1 is a gross indicator of the problem size. This UFP needs to be refined. This is possible, since each parameter (input, output, etc.) has been implicitly assumed to be of average complexity. However, this is rarely true. For example, some input values may be extremely complex, some very simple, etc. In order to take this issue into account, UFP is refined by taking into account the complexities of the parameters of UFP computation (Eq. 3.1). The complexity of each parameter is graded into three broad categories—simple, average, or complex. The weights for the different parameters are determined based on the numerical values shown in Table 3.1. Based on these weights of the parameters, the parameter values in the UFP are refined. For example, rather than each input being computed as four FPs, very simple inputs are computed as three FPs and very complex inputs as six FPs.

Table 3.1: Refinement of Function Point Entities

Type	Simple	Average	Complex
Input(I)	3	4	6
Output (O)	4	5	7
Inquiry (E)	3	4	6
Number of files (F)	7	10	15
Number of interfaces	5	7	10

Step 3: Refine UFP based on complexity of the overall project

In the final step, several factors that can impact the overall project size are considered to refine the UFP computed in step 2. Examples of such project parameters that can influence the project sizes include high transaction rates, response time requirements, scope for reuse, etc. Albrecht identified 14 parameters that can influence the development effort. The list of these parameters have been shown in Table 3.2. Each of these 14 parameters is assigned a value from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI). A technical complexity factor (TCF) for the project is computed and the TCF is multiplied with UFP to yield FP. The TCF expresses the overall impact of the corresponding project parameters on the development effort. TCF is computed as (0.65+0.01*DI). As DI can vary from 0 to 84, TCF can vary from

0.65 to 1.49. Finally, FP is given as the product of UFP and TCF. That is, FP=UFP*TCF.

Table 3.2: Function Point Relative Complexity Adjustment Factors
Requirement for reliable backup and recovery
Requirement for data communication
Extent of distributed processing
Performance requirements
Expected operational environment
Extent of online data entries
Extent of multi-screen or multi-operation online data input
Extent of online updating of master files
Extent of complex inputs, outputs, online queries and files
Extent of complex data processing
Extent that currently developed code can be designed for reuse
Extent of conversion and installation included in the design
Extent of multiple installations in an organisation and variety of customer organisations
Extent of change and focus on ease of use

Example 3.1 Determine the function point measure of the size of the following supermarket software. A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply his/her residence address, telephone number, and the driving license number. Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer. Based on the generated CN, a clerk manually prepares a customer identity card after getting the market manager's signature on it. A customer can present his customer identity card to the check out staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year. Also, it intends to award a 22 caret gold coin to every customer whose purchase exceeded Rs. 10,000. The entries against the CN are reset on the last day of every year after the prize winners' lists are generated. Assume that various project characteristics determining the complexity of software development to be average.

Answer:

Step 1: From an examination of the problem description, we find that there are two inputs, three outputs, two files, and no interfaces. Two files would be required, one for storing the customer details and another for storing the daily purchase records. Now, using equation 3.1, we get:

$$UFP = 2 \times 4 + 3 \times 5 + 1 \times 4 + 10 \times 2 + 0 \times 10 = 47$$

Step 2: All the parameters are of moderate complexity, except the

output parameter of customer registration, in which the only output is the CN value. Consequently, the complexity of the output parameter of the customer registration function can be categorized as simple. By consulting Table 3.1, we find that the value for simple output is given to be 4. The UFP can be refined as follows:

$$UFP = 3 \times 4 + 2 \times 5 + 1 \times 4 + 10 \times 2 + 0 \times 10 = 46$$

Therefore, the UFP will be 46.

Step 3: Since the complexity adjustment factors have average values, therefore the total degrees of influence would be: $DI = 14 \times 4 = 56$

$$TCF = 0.65 + 0.01 + 56 = 1.21$$

Therefore, the adjusted FP=46*1.21=55.66

Feature point metric shortcomings: A major shortcoming of the function point measure is that it does not take into account the algorithmic complexity of a function. That is, the function point metric implicitly assumes that the effort required to design and develop any two different functionalities of the system is the same. But, we know that this is highly unlikely to be true. The effort required to develop any two functionalities may vary widely. For example, in a library automation software, the create-member feature would be much simpler compared to the loan-from-remote-library feature. FP only considers the number of functions that the system supports, without distinguishing the difficulty levels of developing the various functionalities. To overcome this problem, an extension to the function point metric called feature point metric has been proposed.

Feature point metric incorporates algorithm complexity as an extra parameter. This parameter ensures that the computed size using the feature point metric reflects the fact that higher the complexity of a function, the greater the effort required to develop it—therefore, it should have larger size compared to a simpler function.

Critical comments on the function point and feature point metrics

Proponents of function point and feature point metrics claim that these two metrics are language-independent and can be easily computed from the SRS document during project planning stage itself. On the other hand, opponents claim that these metrics are subjective and require a sleight of hand. An example of the subjective nature of the

function point metric can be that the way one groups input and output data items into logically related groups can be very subjective. For example, consider that certain functionality requires the employee name and employee address to be input. It is possible that one can consider both these items as a single unit of data, since after all, these describe a single employee. It is also possible for someone else to consider an employee's address as a single unit of input data and name as another. Such ambiguities leave sufficient scope for debate and keep open the possibility for different project managers to arrive at different function point measures for essentially the same problem.

3.5 PROJECT ESTIMATION TECHNIQUES

Estimation of various project parameters is an important project planning activity. The different parameters of a project that need to be estimated include—project size, effort required to complete the project, project duration, and cost. Accurate estimation of these parameters is important, since these not only help in quoting an appropriate project cost to the customer, but also form the basis for resource planning and scheduling. A large number of estimation techniques have been proposed by researchers. These can broadly be classified into three main categories:

- Empirical estimation techniques
- Heuristic techniques
- Analytical estimation techniques

In the following subsections, we provide an overview of the different categories of estimation techniques.

3.5.1 Empirical Estimation Techniques

Empirical estimation techniques are essentially based on making an educated guess of the project parameters. While using this technique, prior experience with development of similar products is helpful. Although empirical estimation techniques are based on common sense and subjective decisions, over the years, the different activities involved in estimation have been formalised to a large extent. We shall discuss two such formalisations of the basic empirical estimation techniques known as expert judgement and the Delphi techniques in Sections 3.6.1 and 3.6.2 respectively.

3.5.2 Heuristic Techniques

Heuristic techniques assume that the relationships that exist among the different project parameters can be satisfactorily modelled using suitable mathematical expressions. Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the values of the independent parameters in the corresponding mathematical expression. Different heuristic estimation models can be divided into the following two broad categories—single variable and multivariable models.

Single variable estimation models assume that various project characteristic can be predicted based on a single previously estimated basic (independent) characteristic of the software such as its size. A single variable estimation model assumes that the relationship between a parameter to be estimated and the corresponding independent parameter can be characterised by an expression of the following form:

$$\text{Estimated Parameter} = c_1 \square e^{d_1}$$

In the above expression, e represents a characteristic of the software that has already been estimated (independent variable). Estimated P arameter is the dependent parameter (to be estimated). The dependent parameter to be estimated could be effort, project duration, staff size, etc., c_1 and d_1 are constants. The values of the constants c_1 and d_1 are usually determined using data collected from past projects (historical data). The COCOMO model discussed in Section 3.7.1, is an example of a single variable cost estimation model.

A multivariable cost estimation model assumes that a parameter can be predicted based on the values of more than one independent parameter. It takes the following form:

$$\text{Estimated Resource} = c_1 \square p_1^{d_1} + c_2 \square p_2^{d_2} + \ldots$$

where, p_1, p_2, ... are the basic (independent) characteristics of the software already estimated, and c_1, c_2, d_1, d_2, ... are constants. Multivariable estimation models are expected to give more accurate estimates compared to the single variable models, since a project parameter is typically influenced by several independent parameters. The independent parameters influence the dependent parameter to different extents. This is modelled by the different sets of constants c_1 ,

d_1 , c_2 , d_2 , ... Values of these constants are usually determined from an analysis of historical data. The intermediate COCOMO model discussed in Section 3.7.2 can be considered to be an example of a multivariable estimation model.

3.5.3 Analytical Estimation Techniques

Analytical estimation techniques derive the required results starting with certain basic assumptions regarding a project. Unlike empirical and heuristic techniques, analytical techniques do have certain scientific basis. As an example of an analytical technique, we shall discuss the Halstead's software science in Section 3.8. We shall see that starting with a few simple assumptions, Halstead's software science derives some interesting results. Halstead's software science is especially useful for estimating software maintenance efforts. In fact, it outperforms both empirical and heuristic techniques as far as estimating software maintenance efforts is concerned.

3.6 EMPIRICAL ESTIMATION TECHNIQUES

We have already pointed out that empirical estimation techniques have, over the years, been formalised to a certain extent. Yet, these are still essentially euphemisms for pure guess work. These techniques are easy to use and give reasonably accurate estimates. Two popular empirical estimation techniques are—Expert judgement and Delphi estimation techniques. We discuss these two techniques in the following subsection.

3.6.1 Expert Judgement

Expert judgement is a widely used size estimation technique. In this technique, an expert makes an educated guess about the problem size after analysing the problem thoroughly.

Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) that would make up the system and then combines the estimates for the individual modules to arrive at the overall estimate. However, this technique suffers from several shortcomings. The outcome of the expert judgement technique is subject to human errors and individual bias. Also, it is possible that an expert may overlook some factors inadvertently. Further, an expert making an estimate may not have relevant

experience and knowledge of all aspects of a project. For example, he may be conversant with the database and user interface parts, but may not be very knowledgeable about the computer communication part. Due to these factors, the size estimation arrived at by the judgement of a single expert may be far from being accurate.

A more refined form of expert judgement is the estimation made by a group of experts. Chances of errors arising out of issues such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias, and the desire to win contract through overly optimistic estimates is minimised when the estimation is done by a group of experts. However, the estimate made by a group of experts may still exhibit bias. For example, on certain issues the entire group of experts may be biased due to reasons such as those arising out of political or social considerations. Another important shortcoming of the expert judgement technique is that the decision made by a group may be dominated by overly assertive members.

3.6.2 Delphi Cost Estimation

Delphi cost estimation technique tries to overcome some of the shortcomings of the expert judgement approach. Delphi estimation is carried out by a team comprising a group of experts and a co-ordinator. In this approach, the co-ordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate. Estimators complete their individual estimates anonymously and submit them to the co-ordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced their estimations. The co-ordinator prepares the summary of the responses of all the estimators, and also includes any unusual rationale noted by any of the estimators. The prepared summary information is distributed to the estimators. Based on this summary, the estimators re-estimate. This process is iterated for several rounds. However, no discussions among the estimators is allowed during the entire estimation process. The purpose behind this restriction is that if any discussion is allowed among the estimators, then many estimators may easily get influenced by the rationale of an estimator who may be more experienced or senior. After the completion of several iterations of estimations, the co-ordinator takes the responsibility of compiling the results and preparing the final estimate. The Delphi estimation, though consumes more time and effort,

overcomes an important shortcoming of the expert judgement technique in that the results can not unjustly be influenced by overly assertive and senior members.

3.7 COCOMO—A HEURISTIC ESTIMATION TECHNIQUE

COnstructive COst estimation MOdel (COCOMO) was proposed by Boehm [1981]. COCOMO prescribes a three stage process for project estimation. In the first stage, an initial estimate is arrived at. Over the next two stages, the initial estimate is refined to arrive at a more accurate estimate. COCOMO uses both single and multivariable estimation models at different stages of estimation.

The three stages of COCOMO estimation technique are—basic COCOMO, intermediate COCOMO, and complete COCOMO. We discuss these three stages of estimation in the following subsection.

3.7.1 Basic COCOMO Model

Boehm postulated that any software development project can be classified into one of the following three categories based on the development complexity—organic, semidetached, and embedded. Based on the category of a software development project, he gave different sets of formulas to estimate the effort and duration from the size estimate.

Three basic classes of software development projects

In order to classify a project into the identified categories, Boehm requires us to consider not only the characteristics of the product but also those of the development team and development environment. Roughly speaking, the three product development classes correspond to development of application, utility and system software. Normally, data processing programs[1] are considered to be application programs. Compilers, linkers, etc., are utility programs. Operating systems and real-time system programs, etc. are system programs. System programs interact directly with the hardware and programming complexities also arise out of the requirement for meeting timing constraints and concurrent processing of tasks.

Brooks [1975] states that utility programs are roughly three times as difficult to write as application programs and system programs are roughly three times as difficult as utility programs. Thus according to Brooks, the

relative levels of product development complexity for the three categories (application, utility and system programs) of products are 1:3:9.

Boehm's [1981] definitions of organic, semidetached, and embedded software are elaborated as follows:

Organic: We can classify a development project to be of organic type, if the project deals with developing a well-understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.

Semidetached: A development project can be classify to be of semidetached type, if the development team consists of a mixture of experienced and inexperienced staff. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

Embedded: A development project is considered to be of embedded type, if the software being developed is strongly coupled to hardware, or if stringent regulations on the operational procedures exist. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

Observe that in deciding the category of the development project, in addition to considering the characteristics of the product being developed, we need to consider the characteristics of the team members. Thus, a simple data processing program may be classified as semidetached, if the team members are inexperienced in the development of similar products.

For the three product categories, Boehm provides different sets of expressions to predict the effort (in units of person-months) and development time from the size estimation given in kilo lines of source code (KLSC). But, how much effort is one person-month?

> One person month is the effort an individual can typically put in a month. The person-month estimate implicitly takes into account the productivity losses that normally occur due to time lost in holidays, weekly offs, coffee breaks, etc.

What is a person-month?

Person-month (PM) is a popular unit for effort measurement.

> Person-month (PM) is considered to be an appropriate unit for measuring effort, because developers are typically assigned to a project for a certain number of months.

It should be carefully noted that an effort estimation of 100 PM does not imply that 100 persons should work for 1 month. Neither does it imply that 1 person should be employed for 100 months to complete the project. The effort estimation simply denotes the area under the person-month curve (see Figure 3.3) for the project. The plot in Figure 3.3 shows that different number of personnel may work at different points in the project development. The number of personnel working on the project usually increases or decreases by an integral number, resulting in the sharp edges in the plot. We shall elaborate in Section 3.9 how the exact number of persons to work at any time on the product development can be determined from the effort and duration estimates.

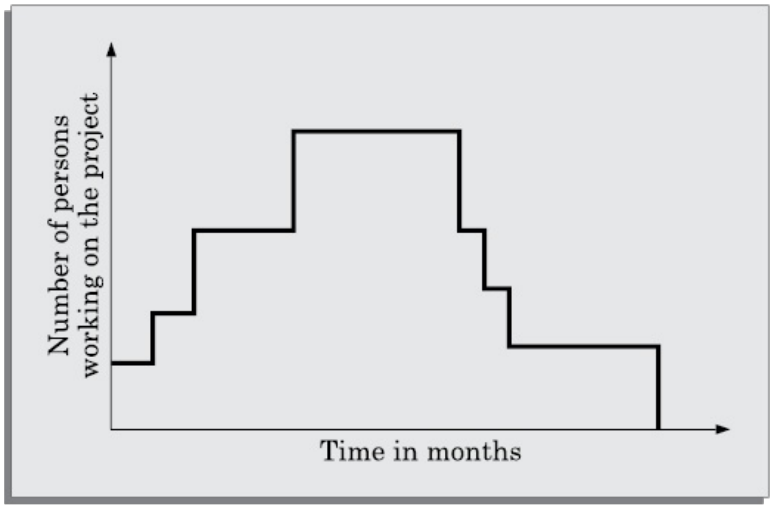

Figure 3.3: Person-month curve.

General form of the COCOMO expressions

The **basic COCOMO model** is a single variable heuristic model that gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by expressions of the following forms:

$$\text{Effort} = a_1 \times (\text{KLOC})^{a_2} \text{ PM}$$

$$\text{Tdev} = b_1 \times (\text{Effort})^{b_2} \text{ months}$$

where,

- KLOC is the estimated size of the software product expressed in Kilo Lines Of Code.
- a_1, a_2, b_1, b_2 are constants for each category of software product.
- Tdev is the estimated time to develop the software, expressed in

months.

- Effort is the total effort required to develop the software product, expressed in person- months (PMs).

According to Boehm, every line of source text should be calculated as one LOC irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines (say n lines), it is considered to be nLOC. The values of a_1, a_2, b_1, b_2 for different categories of products as given by Boehm [1981] are summarised below. He derived these values by examining historical data collected from a large number of actual projects.

Estimation of development effort: For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

Organic : Effort = 2.4(KLOC)1.05 PM

Semi-detached : Effort = 3.0(KLOC)1.12 PM

Embedded : Effort = 3.6(KLOC)1.20 PM

Estimation of development time: For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic : Tdev = 2.5(Effort)0.38 Months

Semi-detached : Tdev = 2.5(Effort)0.35 Months

Embedded : Tdev = 2.5(Effort)0.32 Months

We can gain some insight into the basic COCOMO model, if we plot the estimated effort and duration values for different software sizes. Figure 3.4 shows the plots of estimated effort versus product size for different categories of software products.

Observations from the effort-size plot From Figure 3.4, we can observe that the effort is some what superlinear (that is, slope of the curve>1) in the size of the software product.

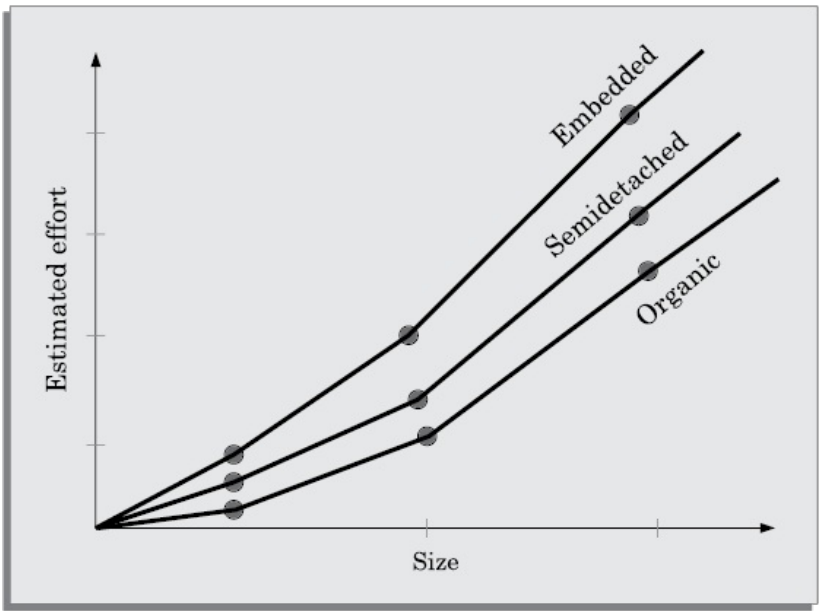

Figure 3.4: Effort versus product size.

This is because the exponent in the effort expression is more than 1. Thus, the effort required to develop a product increases rapidly with project size. However, observe that the increase in effort with size is not as bad as that was portrayed in Chapter 1. The reason for this is that COCOMO assumes that projects are carefully designed and developed by using software engineering principles.

Observations from the development time—size plot

The development time versus the product size in KLOC is plotted in Figure 3.5. From
Figure 3.5, we can observe the following:

- The development time is a sublinear function of the size of the product. That is, when the size of the product increases by two times, the time to develop the product does not double but rises moderately. For example, to develop a product twice as large as a product of size 100KLOC, the increase in duration may only be 20 per cent. It may appear surprising that the duration curve does not increase superlinearly—one would normally expect the curves to behave similar to those in the effort-size plots. This apparent anomaly can be explained by the fact that COCOMO assumes that a project development is carried out not by a single person but by a team of developers.
- From Figure 3.5 we can observe that for a project of any given size, the

development time is roughly the same for all the three categories of products. For example, a 60 KLOC program can be developed in approximately 18 months, regardless of whether it is of organic, semi-detached, or embedded type. (Please verify this using the basic COCOMO formulas discussed in this section). However, according to the COCOMO formulas, embedded programs require much higher effort than either application or utility programs. We can interpret it to mean that there is more scope for parallel activities for system programs than those in utility or application programs.

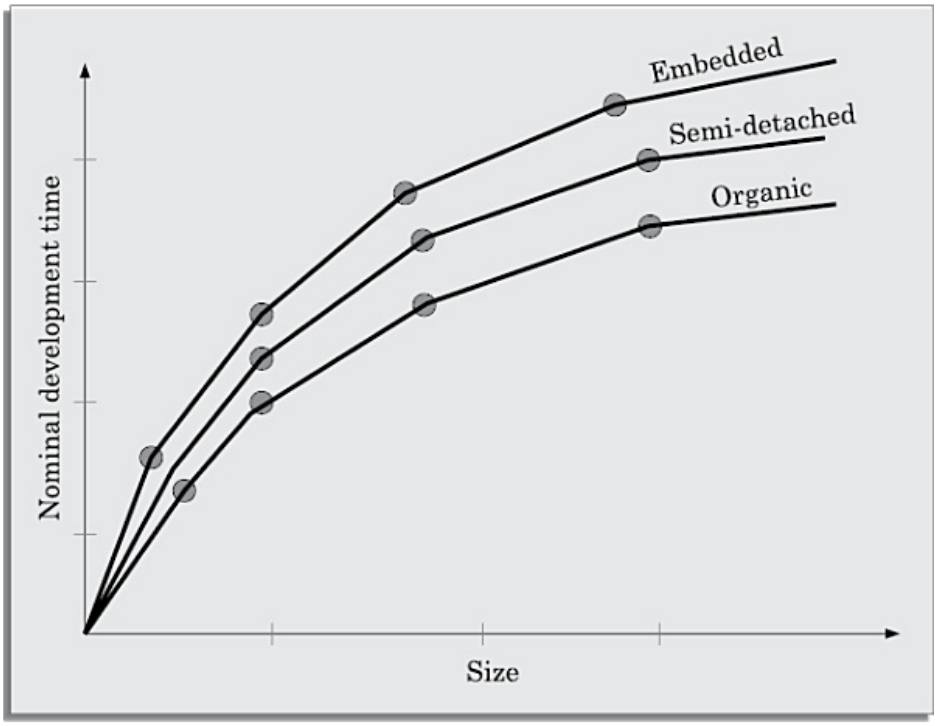

Figure 3.5: Development time versus size.

Cost estimation

From the effort estimation, project cost can be obtained by multiplying the estimated effort (in man-month) by the manpower cost per month. Implicit in this project cost computation is the assumption that the entire project cost is incurred on account of the manpower cost alone. However, in addition to manpower cost, a project would incur several other types of costs which we shall refer to as the overhead costs. The overhead costs would include the costs due to hardware and software required for the project and the company overheads for administration, office space,electricity, etc. Depending on the expected values of the overhead costs, the project manager has to suitably scale up the cost

arrived by using the COCOMO formula.

Implications of effort and duration estimate

An important implicit implication of the COCOMO estimates are that if you try to complete the project in a time shorter than the estimated duration, then the cost will increase drastically. But, if you complete the project over a longer period of time than that estimated, then there is almost no decrease in the estimated cost value. The reasons for this are discussed in Section 3.9. Thus, we can consider that the COCOMO effort and duration values to indicate the following.

> The effort and duration values computed by COCOMO are the values for completing the work in the shortest time without unduly increasing manpower cost.

Let us now elaborate the above statement. When a project team consists of a single member, the member would never be idle for want of work, but the project would take too long to complete. On the other hand, when there are too many members, the project would be completed in much shorter time, but often during the project duration some members would have to idle for want of work.

The project duration is as computed by the COCOMO model, all the developers remain busy with work during the entire development period. Whenever a project is to be completed in a time shorter than the duration estimated by using COCOMO, some idle time on the part of the developers would exist. Such idle times would result in increased development cost. An optimum sized team for a project is one in which any developer any time during development does not sit idle waiting for work, but at the same time consists of as many members as possible to reduce the development time. We can think of the duration given by COCOMO is called the as the optimal duration. It is called optimal duration, if the project is attempted to be completed in any shorter time, then the effort required would rise rapidly. This may appear as a paradox—after all, it is the same product that would be developed, though over a shorter time, then why should the effort required rise rapidly? This can be explained by the fact that for every product at any point during the project development, there is a limit on the number of parallel activities that can meaningfully be identified and carried out. Thus if more number of developers are deployed than the optimal size, some of the developers would have to idle, since at some point in development or other, it would not be possible to assign them any work at all. These idle times

would show up as higher effort and larger cost.

Staff-size estimation

Given the estimations for the project development effort and the nominal development time, can the required staffing level be determined by a simple division of the effort estimation by the duration estimation? The answer is "No". It will be a perfect recipe for project delays and cost overshoot. We examine the staffing problem in more detail in Section 3.9. From the discussions in Section 3.9, it would become clear that the simple division approach to obtain the staff size would be highly improper.

Example 3.2 Assume that the size of an organic type software product has been estimated to be 32,000 lines of source code. Assume that the average salary of a software developer is Rs. 15,000 per month. Determine the effort required to develop the software product, the nominal development time, and the cost to develop the product.

From the basic COCOMO estimation formula for organic software: Effort = $2.4 \times (32)1.05 = 91$ PM

Nominal development time = $2.5 \times (91)0.38 = 14$ months

Staff cost required to develop the product = $91 \times$ Rs. 15, 000 = Rs. 1,465,000

3.7.2 Intermediate COCOMO

The basic COCOMO model assumes that effort and development time are functions of the product size alone. However, a host of other project parameters besides the product size affect the effort as well as the time required to develop the product. For example the effort to develop a product would vary depending upon the sophistication of the development environment.

Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account. The intermediate COCOMO model recognises this fact and refines the initial estimates.

The intermediate COCOMO model refines the initial estimate obtained using the basic COCOMO expressions by scaling the estimate up or down based on the evaluation of a set of attributes of software development.

The intermediate COCOMO model uses a set of 15 cost drivers (multipliers) that are determined based on various attributes of software development. These cost drivers are multiplied with the initial cost and effort estimates (obtained from the basic COCOMO) to appropriately scale those up or down. For example, if modern programming practices are used, the initial estimates are scaled downward by multiplication with a cost driver having a value less than 1. If there are stringent reliability requirements on the software product, the initial estimates are scaled upward. Boehm requires the project manager to rate 15 different parameters for a particular project on a scale of one to three. For each such grading of a project parameter, he has suggested appropriate cost drivers (or multipliers) to refine the initial estimates.

In general, the cost drivers identified by Boehm can be classified as being attributes of the following items:

Product: The characteristics of the product that are considered include the inherent complexity of the product, reliability requirements of the product, etc.

Computer: Characteristics of the computer that are considered include the execution speed required, storage space required, etc.

Personnel: The attributes of development personnel that are considered include the experience level of personnel, their programming capability, analysis capability, etc.

Development environment: Development environment attributes capture the development facilities available to the developers. An important parameter that is considered is the sophistication of the automation (CASE) tools used for software development.

We have discussed only the basic ideas behind the intermediate COCOMO model. A detailed discussion on the intermediate COCOMO model are beyond the scope of this book and the interested reader may refer [Boehm81].

3.7.3 Complete COCOMO

A major shortcoming of both the basic and the intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large systems are made up of several smaller sub-systems. These sub-systems often have widely different characteristics. For example, some sub-systems may be considered as organic type, some semidetached, and some even embedded. Not only may the inherent development complexity of the

subsystems be different, but for some subsystem the reliability requirements may be high, for some the development team might have no previous experience of similar development, and so on.

The complete COCOMO model considers these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual sub-systems.

In other words, the cost to develop each sub-system is estimated separately, and the complete system cost is determined as the subsystem costs. This approach reduces the margin of error in the final estimate.

Let us consider the following development project as an example application of the complete COCOMO model. A distributed management information system (MIS) product for an organisation having offices at several places across the country can have the following sub-component:

- Database part
- Graphical user interface (GUI) part
- Communication part

Of these, the communication part can be considered as embedded software. The database part could be semi-detached software, and the GUI part organic software. The costs for these three components can be estimated separately, and summed up to give the overall cost of the system.

To further improve the accuracy of the results, the different parameter values of the model can be fine-tuned and validated against an organisation's historical project database to obtain more accurate estimations. Estimation models such as COCOMO are not totally accurate and lack a full scientific justification. Still, software cost estimation models such as COCOMO are required for an engineering approach to software project management. Companies consider computed cost estimates to be satisfactory, if these are within about 80 per cent of the final cost. Although these estimates are gross approximations—without such models, one has only subjective judgements to rely on.

3.7.4 COCOMO 2

Since the time that COCOMO estimation model was proposed in the early 1980s, the software development paradigms as well as the characteristics of development projects have undergone a sea change. The present day software projects are much larger in size and reuse of existing software to develop new products has become pervasive. For example, component-based development and service-oriented

architectures (SoA) have become very popular (discussed in Chapter 15). New life cycle models and development paradigms are being deployed for web-based and component-based software. During the 1980s rarely any program was interactive, and graphical user interfaces were almost non-existent. On the other hand, the present day software products are highly interactive and support elaborate graphical user interface. Effort spent on developing the GUI part is often as much as the effort spent on developing the actual functionality of the software. To make COCOMO suitable in the changed scenario, Boehm proposed COCOMO 2 [Boehm95] in 1995.

COCOMO 2 provides three models to arrive at increasingly accurate cost estimations. These can be used to estimate project costs at different phases of the software product. As the project progresses, these models can be applied at the different stages of the same project.

Application composition model: This model as the name suggests, can be used to estimate the cost for prototype development. We had already discussed in Chapter 2 that a prototype is usually developed to resolve user interface issues.

Early design model: This supports estimation of cost at the architectural design stage.

Post-architecture model: This provides cost estimation during detailed design and coding stages.

The post-architectural model can be considered as an update of the original COCOMO. The other two models help consider the following two factors. Now a days every software is interactive and GUI-driven. GUI development constitutes a significant part of the overall development effort. The second factor concerns several issues that affect productivity such as the extent of reuse. We briefly discuss these three models in the following.

Application composition model

The application composition model is based on counting the number of screens, reports, and modules (components). Each of these components is considered to be an object (this has nothing to do with the concept of objects in the object-oriented paradigm). These are used to compute the object points of the application.

Effort is estimated in the application composition model as follows:

1. Estimate the number of screens, reports, and modules (components)

from an analysis of the SRS document.

2. Determine the complexity level of each screen and report, and rate these as either simple, medium, or difficult. The complexity of a screen or a report is determined by the number of tables and views it contains.
3. Use the weight values in Table 3.3 to 3.5.

The weights have been designed to correspond to the amount of effort required to implement an instance of an object at the assigned complexity class.

Table 3.3: SCREEN Complexity Assignments for the Data Tables

Number of views	Tables < 4	Tables < 8	Tables ≥ 8
< 3	Simple	Simple	Medium
3–7	Simple	Medium	Difficult
>8	Medium	Difficult	Difficult

Table 3.4: Report Complexity Assignments for the Data Tables

Number of views	Tables < 4	Tables < 8	Tables ≥ 8
0 or 1	Simple	Simple	Medium
2 or 3	Simple	Medium	Difficult
4 or more	Medium	Difficult	Difficult

4. Add all the assigned complexity values for the object instances together to obtain the object points.

Table 3.5: Table of Complexity Weights for Each Class for Each Object Type

Object type	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component	—	—	10

5. Estimate percentage of reuse expected in the system. Note that reuse refers to the amount of pre-developed software that will be used within the system. Then, evaluate New Object-Point count (NOP) as follows,

$$NOP = \frac{(\text{Object-Points})(100 - \%\text{ of reuse})}{100}$$

6. Determine the productivity using Table 3.6. The productivi ty depends on

the experience of the developers as well as the maturity of the CASE environment used.

7. Finally, the estimated effort in person-months is computed as E = NOP/PROD.

Table 3.6: Productivity Table

Developers' experience	Very low	Low	Nominal	High	Very high
CASE maturity	Very low	Low	Nominal	High	Very high
PRODUCTIVITY	4	7	13	25	50

Early design model

The unadjusted function points (UFP) are counted and converted to source lines of code (SLOP). In a typical programming environment, each UFP would correspond to about 128 lines of C, 29 lines of C++, or 320 lines of assembly code. Of course, the conversion from UFP to LOC is environment specific, and depends on factors such as extent of reusable libraries supported. Seven cost drivers that characterise the post-architecture model are used. These are rated on a seven points scale. The cost drivers include product reliability and complexity, the extent of reuse, platform sophistication, personnel experience, CASE support, and schedule.

The effort is calculated using the following formula:

$$\text{Effort} = \text{K SLOC} \times \square_{i} \text{ cost driver}_i$$

Post-architecture model

The effort is calculated using the following formula, which is similar to the original COCOMO model.

$$\text{Effort} = a \times \text{K SLOC}^{b} \times \square_{i} \text{ cost driver}_i$$

The post-architecture model differs from the original COCOMO model in the choice of the set of cost drivers and the range of values of the exponent b. The exponent b can take values in the range of 1.01 to 1.26. The details of the COCOMO 2 model, and the exact values of b and the cost drivers can be found in [Boehm 97].

3.8 HALSTEAD'S SOFTWARE SCIENCE—AN ANALYTICAL TECHNIQUE

Halstead's software science[2] is an analytical technique to measure size,

development effort, and development cost of software products. Halstead used a few primitive program parameters to develop the expressions for over all program length, potential minimum volume, actual volume, language level, effort, and development time.

For a given program, let:

- h_1 be the number of unique operators used in the program,
- h_2 be the number of unique operands used in the program,
- N_1 be the total number of operators used in the program,
- N_2 be the total number of operands used in the program.

Although the terms operators and operands have intuitive meanings, a precise definition of these terms is needed to avoid ambiguities. But, unfortunately we would not be able to provide a precise definition of these two terms. There is no general agreement among researchers on what is the most meaningful way to define the operators and operands for different programming languages. However, a few general guidelines regarding identification of operators and operands for any programming language can be provided. For instance, assignment, arithmetic, and logical operators are usually counted as operators. A pair of parentheses, as well as a block begin —block end pair, are considered as single operators. A label is considered to be an operator, if it is used as the target of a GOTO statement. The constructs `if ... then ... else ... endif` and `a while ... do` are considered as single operators. A sequence (statement termination) operator ';' is considered as a single operator. Subroutine declarations and variable declarations comprise the operands. Function name in a function call statement is considered as an operator, and the arguments of the function call are considered as operands. However, the parameter list of a function in the function declaration statement is not considered as operands. We list below what we consider to be the set of operators and operands for the ANSI C language. However, it should be realised that there is considerable disagreement among various researchers in this regard.

Operators and Operands for the ANSI C language

The following is a suggested list of operators for the ANSI C language:

```
( [ . , -> * + - ~ ! ++ -- * / % + - << >> < > <= >= !=
== & ^ | && || = *= /= %= += -= <<= >>= &= ^= |= : ? { ;
```

```
CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE
BREAK RETURN and a function name in a function call
```

Operands are those variables and constants which are being used with operators in expressions. Note that variable names appearing in declarations are not considered as operands.

Example 3.3 Consider the expression a = &b; a, b are the operands and =, & are the operators.

Example 3.4 The function name in a function definition is not counted as an operator.

```
int func ( int a, int b )
{
      . . .
}
```

For the above example code, the operators are: {}, ( ) We do not consider func, a, and b as operands, since these are part of the function definition.

Example 3.5 Consider the function call statement: func (a, b);. In this, func ',' and ; are considered as operators and variables a, b are treated as operands.

3.8.1 Length and Vocabulary

The length of a program as defined by Halstead, quantifies total usage of all operators and operands in the program. Thus, length N = N_1 + N_2. Halstead's definition of the length of the program as the total number of operators and operands roughly agrees with the intuitive notion of the program length as the total number of tokens used in the program.

The program vocabulary is the number of unique operators and operands used in the program. Thus, program vocabulary $h = h_1 + h_2$.

3.8.2 Program Volume

The length of a program (i.e., the total number of operators and operands used in the code) depends on the choice of the operators and operands used. In other words, for the same programming problem, the length would depend on the programming style. This type of dependency would produce different measures of length for essentially the same problem when different programming languages are used.

Thus, while expressing program size, the programming language used must be taken into consideration:

$$V = N \log_2 h$$

Let us try to understand the important idea behind this expression. Intuitively, the program volume V is the minimum number of bits needed to encode the program. In fact, to represent h different identifiers uniquely, we need at least $\log_2 h$ bits (where h is the program vocabulary). In this scheme, we need $N \log_2 h$ bits to store a program of length N. Therefore, the volume V represents the size of the program by approximately compensating for the effect of the programming language used.

3.8.3 Potential Minimum Volume

The potential minimum volume V* is defined as the volume of the most succinct program in which a problem can be coded. The minimum volume is obtained when the program can be expressed using a single source code instruction, say a function call like `foo()`;. In other words, the volume is bound from below due to the fact that a program would have at least two operators and no less than the requisite number of operands. Note that the operands are the input and output data items.

Thus, if an algorithm operates on input and output data $d_1, d_2, \ldots d_n$, the most succinct program would be $f(d_1, d_2, \ldots, d_n)$; for which, $h_1 = 2$, $h_2 = n$. Therefore, $V^* = (2 + h_2) \log_2 (2 + h_2)$.

The program level L is given by $L = V^*/V$. The concept of program level L has been introduced in an attempt to measure the level of abstraction provided by the programming language. Using this definition, languages can be ranked into levels that also appear intuitively correct.

The above result implies that the higher the level of a language, the less effort it takes to develop a program using that language. This result agrees with the intuitive notion that it takes more effort to develop a program in assembly language than to develop a program in a high-level language to solve a problem.

3.8.4 Effort and Time

The effort required to develop a program can be obtained by dividing the program volume with the level of the programming language used to

develop the code. Thus, effort E = V /L, where E is the number of mental discriminations required to implement the program and also the effort required to read and understand the program. Thus, the programming effort E = V^2/V^* (since L = V*/V) varies as the square of the volume. Experience shows that E is well correlated to the effort needed for maintenance of an existing program.

The programmer's time T = E/S, where S is the speed of mental discriminations. The value of S has been empirically developed from psychological reasoning, and its recommended value for programming applications is 18.

3.8.5 Length Estimation

Even though the length of a program can be found by calculating the total number of operators and operands in a program, Halstead suggests a way to determine the length of a program using the number of unique operators and operands used in the program. Using this method, the program parameters such as length, volume, cost, effort, etc., can be determined even before the start of any programming activity. His method is summarised below.

Halstead assumed that it is quite unlikely that a program has several identical parts— in formal language terminology identical substrings—of length greater than h(h being the program vocabulary). In fact, once a piece of code occurs identically at several places, it is usually made into a procedure or a function. Thus, we can safely assume that any program of length N consists of N/h unique strings of length h. Now, it is a standard combinatorial result that for any given alphabet of size K, there are exactly K^r different strings of length r. Thus,

$$\frac{N}{\eta} \leq \eta^{\eta}$$

or

$$N \leq \eta^{\eta+1}$$

Since operators and operands usually alternate in a program, we can further refine the upper bound into N ≤ $hh_1^{h1}\ h_2^{h3}$. Also, N must include not only the ordered set of N elements, but it should also include all possible subsets of that ordered set, i.e. the power set of N strings

(This particular reasoning of Halstead is hard to justify!).

Therefore,

$$2^N = \eta\eta_1^{\eta_1}\eta_2^{\eta_2}$$

or, taking logarithm on both sides,

$$N = \log_2 \eta + log_2(\eta_1^{\eta_1}\eta_2^{\eta_2})$$

So, we get,

$$N = \log_2(\eta_1^{\eta_1}\eta_2^{\eta_2}) \quad \text{(approximately, by ignoring } \log_2 \eta)$$

or,

$$N = log_2\eta_1^{\eta_1} + log_2\eta_2^{\eta_2}$$
$$= \eta_1 log_2\eta_1 + \eta_2 log_2\eta_2$$

Experimental evidence gathered from the analysis of a large number of programs suggests that the computed and actual lengths match very closely. However, the results may be inaccurate when small programs are considered individually.

Example 3.6 Let us consider the following C program:

```
main()
{
  int a,b,c,avg;
  scanf("%d %d %d",&a,&b,&c);
  avg=(a+b+c)/3;
  printf("avg= %d",avg);
}
```

The unique operators are: `main, (), {}, int, scanf, &, “,”, “;”, =, +, /, printf`

The unique operands are: `a,b,c,&a,&b,&c,a+b+c,avg,3,”%d %d %d”, “avg=%d”`

Therefore,

$$\eta_1 = 12, \eta_2 = 11$$
$$\text{Estimated Length} = (12 * \log 12 + 11 * \log 11)$$
$$= (12 * 3.58 + 11 * 3.45) = (43 + 38) = 81$$
$$\text{Volume} = \text{Length} * \log(23) = 81 * 4.52 = 366$$

In conclusion, Halstead’s theory tries to provide a formal definition and quantification of such qualitative attributes as program complexity, ease of understanding, and the level of abstraction based on some low-level parameters such as the number of operands, and operators appearing in the program. Halstead’s software science provides gross estimates of properties

of a large collection of software, but extends to individual cases rather inaccurately.

3.9 STAFFING LEVEL ESTIMATION

Once the effort required to complete a software project has been estimated, the staffing requirement for the project can be determined. Putnam was the first to study the problem of determining a proper staffing pattern for software projects. He extended the classical work of Norden who had earlier investigated the staffing pattern of general research and development (R&D) type of projects. In order to appreciate the uniqueness of the staffing pattern that is desirable for software projects, we must first understand both Norden's and Putnam's results.

3.9.1 Norden's Work

Norden studied the staffing patterns of several R&D projects. He found that the staffing pattern of R&D type of projects is very different from that of manufacturing or sales. In a sales outlet, the number of sales staff does not usually vary with time. For example, in a supermarket the number of sales personnel would depend on the number of sales counters and would be approximately constant over time. However, the staffing pattern of R&D type of projects needs to change over time. At the start of an R&D project, the activities of the project are planned and initial investigations are made. During this time, the manpower requirements are low. As the project progresses, the manpower requirement increases, until it reaches a peak. Thereafter, the manpower requirement gradually diminishes.

Norden concluded that the staffing pattern for any R&D project starting from a low level, increases until it reaches a peak value. It then starts to diminish. This pattern can be approximated by the Rayleigh distribution curve (see Figure 3.6).

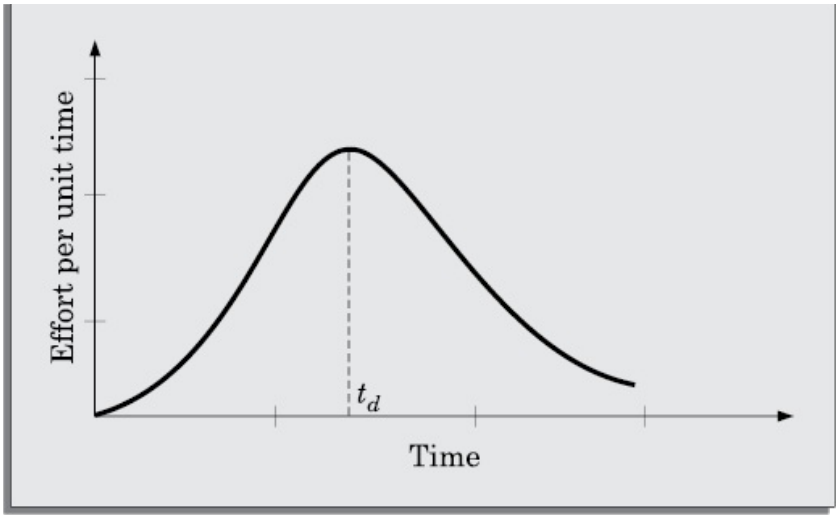

Figure 3.6: Rayleigh curve.

Norden represented the Rayleigh curve by the following equation:

$$E = \frac{K}{{t_d}^2} * t * e^{\frac{-t^2}{2{t_d}^2}}$$

where, E is the effort required at time t. E is an indication of the number of developers (or the staffing level) at any particular time during the duration of the project, K is the area under the curve, and td is the time at which the curve attains its maximum value. It must be remembered that the results of Norden are applicable to general R&D projects and were not meant to model the staffing pattern of software development projects.

3.9.2 Putnam's Work

Putnam studied the problem of staffing of software projects and found that the staffing pattern for software development projects has characteristics very similar to any other R&D projects. Only a small number of developers are needed at the beginning of a project to carry out the planning and specification tasks. As the project progresses and more detailed work is performed, the number of developers increases and reaches a peak during product testing. After implementation and unit testing, the number of project staff falls.

Putnam found that the Rayleigh-Norden curve can be adapted to relate the number of delivered lines of code to the effort and the time required to develop the product. By analysing a large number of defence projects, Putnam derived the following expression:

$$L = C_k K^{1/3} t_d^{4/3} \qquad (3.2)$$

where the different terms are as follows:

- K is the total effort expended (in PM) in the product development and L is the product size in KLOC.
- t_d corresponds to the time of system and integration and testing. Therefore, t_d can be approximately considered as the time required to develop the software.
- Ck is the state of technology constant and reflects constraints that impede the progress of the programmer. Typical values of $C_k = 2$ for poor development environment (no methodology, poor documentation, and review, etc.), $C_k = 8$ for good software development environment (software engineering principles are adhered to), $C_k = 11$ for an excellent environment (in addition to following software engineering principles, automated tools and techniques are used). The exact value of C_k for a specific project can be computed from historical data of the organisation developing it.

Putnam suggested that optimal staff build-up on a project should follow the Rayleigh curve.

For efficient resource utilisation as well as project completion over optimal duration, starting from a small number of developers, there should be a staff build-up and after a peak size has been achieved, staff reduction is required. However, the staff build-up should not be carried out in large installments. The team size should either be increased or decreased slowly whenever required to match the Rayleigh-Norden curve.

Experience reports indicate that a very rapid build up of project staff any time during the project development correlates with schedule slippage.

It should be clear that a constant level of manpower throughout the project duration would lead to wastage of effort and as a result would increase both the time and effort required to develop the product. If a constant number of developers are used over all the phases of a project, some phases would be overstaffed and the other phases would be understaffed causing inefficient use of manpower, leading to schedule slippage and increase in cost.

If we examine the Rayleigh curve, we can see that approximately 40 per cent of the area under the Rayleigh curve is to the left of t_d and 60 per cent area is to the right of t_d. This has been verified mathematically by integrating the expression provided by Putnam. This implies that the effort

required to develop the product to its maintenance effort is approximately in 40:60 ratio. We had already pointed out in Chapter 2 that this is an expected pattern of distribution of effort between the development and maintenance of a product.

Effect of schedule change on cost according to Putnom method

Putnam's method (Eq. 3.2) can be used to study the effect of changing the duration of a project from that computed by the COCOMO model. By using the Putnam's expression Eq. (3.2):

$$K = \frac{L^3}{(C_k^3 t_d^4)}$$

or,

$$K = \frac{C}{t_d{}^4}$$

For the same product size, $C = \frac{L^3}{C_k^3}$ is a constant.

Or,

$$\frac{K_1}{K_2} = \frac{t_{d_2}^4}{t_{d_1}^4} \quad (3.3)$$

From this expression, it can easily be observed that when the schedule of a project is compressed, the required effort increases in proportion to the fourth power of the degree of compression. It means that a relatively small compression in delivery schedule can result in substantial penalty on human effort. For example, if the estimated development time using COCOMO formulas is 1 year, then in order to develop the product in 6 months, the total effort required (and hence the project cost) increases 16 times.

Example 3.7 The nominal effort and duration of a project have been estimated to be 1000PM and 15 months. The project cost has been negotiated to be Rs. 200,000,000. The needs the product to be developed and delivered in 12 month time. What should be the new cost to be negotiated?

Answer: The project can be classified as a large project. Therefore, the new cost to be negotiated can be given by the Putnam's formula: new cost = Rs. 200, 000, 000 × $(15/12)^4$ = Rs. 488,281,250.

Why does project cost increase when schedule is compressed?

It is a common intuition that the effort required to develop a product should not depend on the time over which it is developed. Why then does the effort requirement increase so much (16 times as per Eq. 3.3) when the schedule is compressed by 50 per cent? After all, it is the same product that is being developed? The answer to this can be the following.

The extra effort can be attributed to the idle times of the developers waiting for work. The project manager recruits large number of developers hoping to complete the project early, but it becomes very difficult to keep those additional developers continuously occupied with work. Implicit in the schedule and cost estimation arrived at using the COCOMO model, is the fact that all developers can be continuously assigned work. However, when more number of developers are hired to decrease the duration, it becomes to keep all developers busy all the time. After all, the activities in the project which can be carried out simultaneously are restricted. As a corollary of this observation, it can be remarked that benefits can be gained by using fewer people over a somewhat longer time span to accomplish the same objective. Thus, the Putnam's model indicates an extreme manpower penalty for schedule compression and an extreme reward for expanding the schedule. However, increasing the development time beyond the duration computed by COCOMO has been found to be not very helpful in reducing the cost.

The Putnam's estimation model works reasonably well for very large systems, but seriously overestimates the required effort on medium and small systems. This is also corroborated by Boehm[Boehm 81].

Boehm states that there is a limit beyond which a software project cannot reduce its schedule by buying any more personnel or equipment.

This limit occurs roughly at 75 per cent of the nominal time estimate for small and medium sized projects. Thus, if a project manager accepts a customer demand to compress the development schedule of a typical project (medium or small project) by more than 25 per cent, he is very unlikely to succeed. The main reason being that every project has only a limited amount of activities which can be carried out in parallel and the sequential activities cannot be speeded up by hiring any number of additional developers.

3.9.3 Jensen's Model

Jensen model [Jensen 84] is very similar to Putnam model. However, it attempts to soften the effect of schedule compression on effort to make it

applicable to smaller and medium sized projects. Jensen proposed the equation:

$$L = C_{te} t_d K^{1/2}$$

or,

$$\frac{K_1}{K_2} = \frac{t_{d_2}^2}{t_{d_1}^2}$$

where, Cte is the effective technology constant, td is the time to develop the software, and K is the effort needed to develop the software.

In contrast to the Putnam's model, Jensen's model considers the increase in effort (and cost) requirement to be proportional to the square of the degree of compression.

3.10 SCHEDULING

Scheduling the project tasks is an important project planning activity.

The scheduling problem, in essence, consists of deciding which tasks would be taken up when and by whom.

Once a schedule has been worked out and the project gets underway, the project manager monitors the timely completion of the tasks and takes any corrective action that may be necessary whenever there is a chance of schedule slippage. In order to schedule the project activities, a software project manager needs to do the following:

1. Identify all the major activities that need to be carried out to complete the project.
2. Break down each activity into tasks.
3. Determine the dependency among different tasks.
4. Establish the estimates for the time durations necessary to complete the tasks.
5. Represent the information in the form of an activity network.
6. Determine task starting and ending dates from the information represented in the activity network.
7. Determine the critical path. A critical path is a chain of tasks that determines the duration of the project.
8. Allocate resources to tasks.

The first step in scheduling a software project involves identifying all the

activities necessary to complete the project. A good knowledge of the intricacies of the project and the development process helps the managers to effectively identify the important activities of the project. Next, the activities are broken down into a logical set of smaller activities (subactivities). The smallest subactivities are called tasks which are assigned to different developers.

The smallest unit of work activities that are subject to management planning and control are called tasks.

A project manager breakdowns the tasks systematically by using the work breakdown structure technique discussed in Section 3.10.1.

After the project manager has broken down the activities into tasks, he has to find the dependency among the tasks. Dependency among the different tasks determines the order in which the different tasks would be carried out. If a task A requires the results of another task B, then task A must be scheduled after task B and A is said to be dependent on B. In general, the task dependencies define a partial ordering among tasks. That is, each tasks may precede a subset of other tasks, but some tasks might not have any precedence ordering defined between them (called concurrent task). The dependency among the activities are represented in the form of an activity network discussed in Section 3.10.2.

Once the activity network representation has been worked out, resources are allocated to each activity. Resource allocation is typically done using a Gantt chart. After resource allocation is done, a project evaluation and review technique (PERT) chart representation is developed. The PERT chart representation is useful to a project manager to carry out project monitoring and control. Let us now discuss the work break down structure, activity network, Gantt and PERT charts.

3.10.1 Work Breakdown Structure

Work breakdown structure (WBS) is used to recursively decompose a given set of activities into smaller activities.

> Tasks are the lowest level work activities in a WBS hierarchy. They also form the basic units of work that are allocated to the developer and scheduled

First, let us understand why it is necessary to break down project activities into tasks. Once project activities have been decomposed into a set of tasks using WBS, the time frame when each activity is to be performed is to be determined. The end of each important activity is called a milestone. The

project manager tracks the progress of a project by monitoring the timely completion of the milestones. If he observes that some milestones start getting delayed, he carefully monitors and controls the progress of the tasks, so that the overall deadline can still be met.

WBS provides a notation for representing the activities, sub-activities, and tasks needed to be carried out in order to solve a problem. Each of these is represented using a rectangle (see Figure 3.7). The root of the tree is labelled by the project name. Each node of the tree is broken down into smaller activities that are made the children of the node. To decompose an activity to a sub-activity, a good knowledge of the activity can be useful. Figure 3.7 represents the WBS of a management information system (MIS) software.

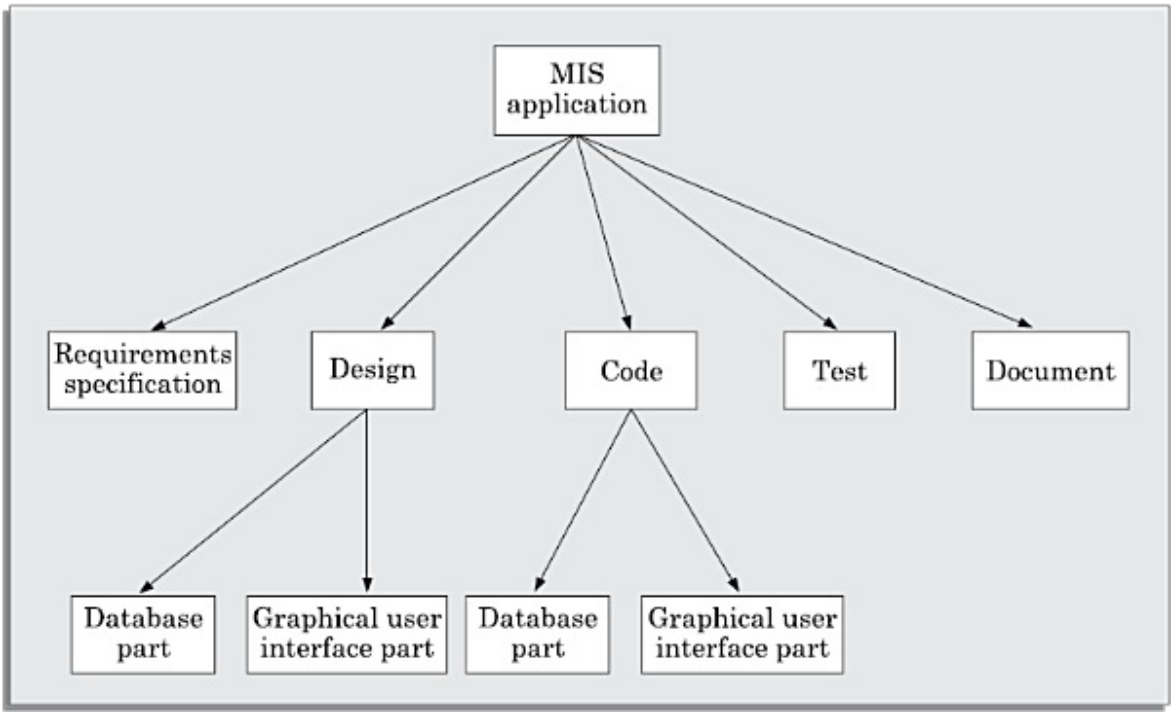

Figure 3.7: Work breakdown structure of an MIS problem.

How long to decompose?

The decomposition of the activities is carried out until any of the following is satisfied:

- A leaf-level subactivity (a task) requires approximately two weeks to develop.
- Hidden complexities are exposed, so that the job to be done is understood and can be assigned as a unit of work to one of the developers.

- Opportunities for reuse of existing software components is identified.

Breaking down tasks to too coarse levels versus too fine levels

Let us first investigate the implications of carrying out the decompositions to very fine levels versus leaving the decomposition at rather coarse grained. It is necessary to breakdown large activities into many smaller tasks, because smaller tasks allow milestones to be placed at shorter distances. This allows close monitoring of the project and corrective actions can be initiated as soon as any problems are noticed. However, very fine subdivision means that a disproportionate amount of time must be spent on preparing and revising various charts.

Let us now investigate the implications of carrying out decompositions to very coarse levels versus decomposing to very fine levels. When the granularity of the tasks is several months, by the time a problem (schedule delay) is noticed and corrective actions are initiated, it may be too late for the project to recover. On the other hand, if the tasks are decomposed into very small granularity (one or two days each), then the milestones get too closely spaced. This would require frequent monitoring of the project status and entail frequent revisions to the plan document. This becomes a high overhead on the project manager and his effectiveness in project monitoring and control gets reduced.

While breaking down an activity into smaller tasks, a manager often has to make some hard decisions. If a activity is broken down into a large number of very small sub-activities, these can be distributed to a larger number of developers. If the task ordering permits that solutions to these can be carried out independently (parallelly), it becomes possible to develop the product faster (with the help of additional manpower of course!). Therefore, to be able to complete a project in the least amount of time, the manager needs to break large tasks into smaller ones, expecting to find more parallelism. However, it is not useful to subdivide tasks into units which take less than a week or two to execute.

Example 3.8 Consider a project for development of a management information system (MIS). The project manager has identified the main development activities to be, requirements specification, design, code, test, and document. decompose the activities into tasks using work breakdown structure technique.

Answer: Based on the manager's domain knowledge, he could arrive at the work breakdown structure shown in Figure 3.8.

3.10.2 Activity Networks

An activity network shows the different activities making up a project, their estimated durations, and their interdependencies. Two equivalent representations for activity networks are possible and are in use:

Activity on Node (AoN): In this representation, each activity is represented by a rectangular (some use circular) node and the duration of the activity is shown alongside each task in the node. The inter-task dependencies are shown using directional edges (see Figure 3.8).

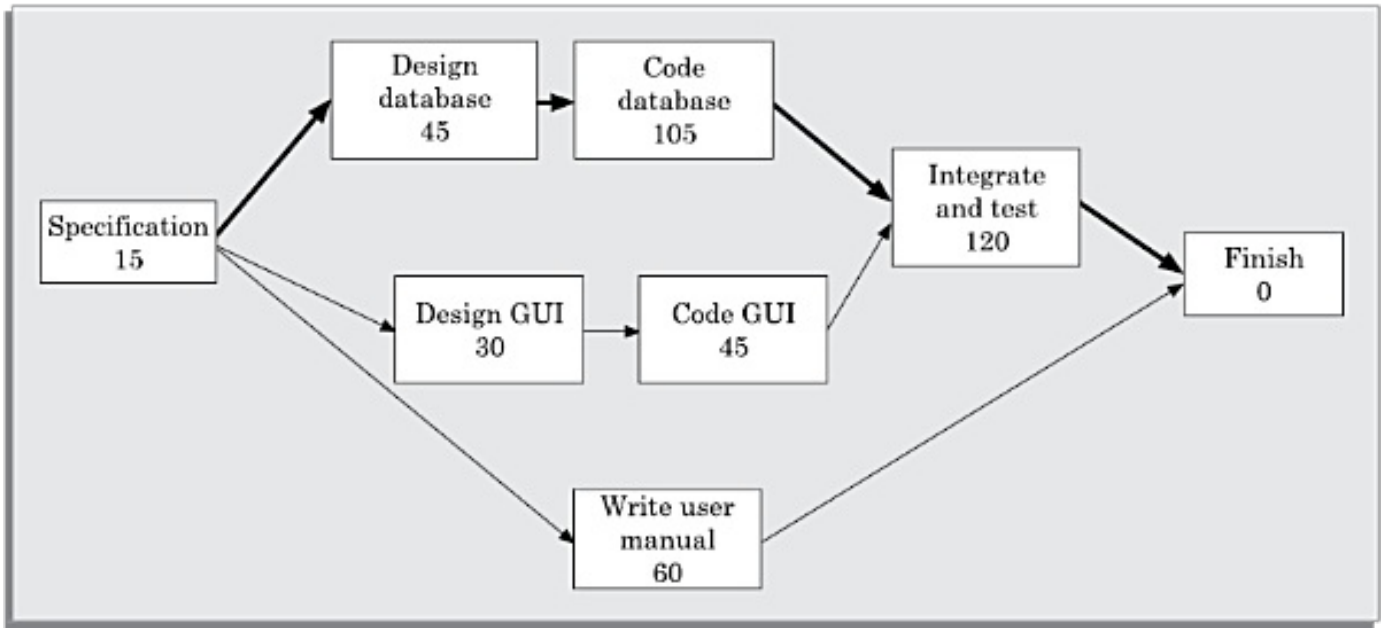

Figure 3.8: Activity network representation of the MIS problem.

Activity on Edge (AoE): In this representation tasks are associated with the edges. The edges are also annotated with the task duration. The nodes in the graph represent project milestones.

Activity networks were originally represented using activity on edge (AoE) representation. However, later activity on node (AoN) has become popular since this representation is easier to understand and revise.

Managers can estimate the time durations for the different tasks in several ways. One possibility is that they can empirically assign durations to different tasks. This however may not be such a good idea, because software developers often resent such unilateral decisions. However, some managers prefer to estimate the time for various activities themselves. They believe that an aggressive schedule would motivate the developers to do a better

and faster job. On the other hand, careful experiments have shown that unrealistically aggressive schedules not only cause developers to compromise on intangible quality aspects, but also cause greater schedule delays compared to the other approaches. A possible alternative is to let each developer himself estimate the time for an activity he would be assigned to. This approach can help to accurately estimate the task durations without creating undue schedule pressures.

Example 3.9: Determine the Activity network representation for the MIS development project of Example 3.7. Assume that the manager has determined the tasks to be represented from the work breakdown structure of Figure 3.7, and has determined the durations and dependencies for each task as shown in Table 3.7.

Answer: The activity network representation has been shown in Figure 3.8.

Table 3.7: Project Parameters Computed from Activity Network

Task Number	Task	Duration	Dependent on Tasks
T1	Specification	15	–
T2	Design database	45	T 1
T3	Design GUI	30	T 1
T4	Code database	105	T 2
T5	Code GUI part	45	T 3
T6	Integrate and test	120	T 4 and T 5
T7	Write user manual	60	T 1

3.10.3 Critical Path Method (CPM)

CPM and PERT are operation research techniques that were developed in the late 1950s. Since then, they have remained extremely popular among project managers. Of late, these two techniques have got merged and many project management tools support them as CPM/PERT. However, we point out the fundamental differences between the two and discuss CPM in this subsection and PERT in the next subsection.

A path in the activity network graph is any set of consecutive nodes and edges in this graph from the starting node to the last node. A critical path consists of a set of dependent tasks that need to be performed in a sequence and which together take the longest time to complete.

A critical task is one with a zero slack time. A path from the start node to the finish node containing only critical tasks is called a critical path.

CPM is an algorithmic approach to determine the critical paths and slack times for tasks not on the critical paths involves calculating the following quantities:

Minimum time (MT): It is the minimum time required to complete the project. It is computed by determining the maximum of all paths from start to finish.

Earliest start (ES): It is the time of a task is the maximum of all paths from the start to this task. The ES for a task is the ES of the previous task plus the duration of the preceding task.

Latest start time (LST): It is the difference between MT and the maximum of all paths from this task to the finish. The LST can be computed by subtracting the duration of the subsequent task from the LST of the subsequent task.

Earliest finish time (EF): The EF for a task is the sum of the earliest start time of the task and the duration of the task.

Latest finish (LF): LF indicates the latest time by which a task can finish without affecting the final completion time of the project. A task completing beyond its LF would cause project delay. LF of a task can be obtained by subtracting maximum of all paths from this task to finish from MT.

Slack time (ST): The slack time (or float time) is the total time that a task may be delayed before it will affect the end time of the project. The slack time indicates the ”flexibility” in starting and completion of tasks. ST for a task is LS-ES and can equivalently be written as LF-EF.

Example 3.10 Use the Activity network of Figure 3.8 to determine the ES and EF for every task for the MIS problem of Example 3.7.

Answer: The activity network with computed ES and EF values has been shown in Figure 3.9.

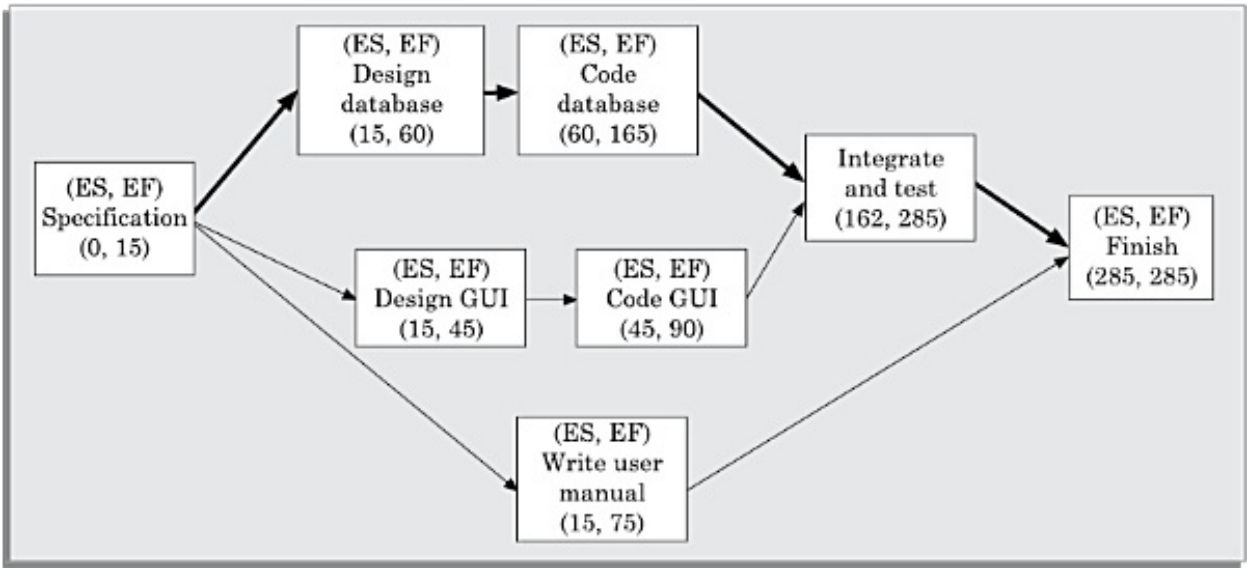

Figure 3.9: AoN for MIS problem with (ES,EF).

Example 3.11 Use the Activity network of Figure 3.9 to determine the LS and LF for every task for the MIS problem of Example 3.7.

Answer: The activity network with computed LS and LF values has been shown in Figure 3.10.

CPM can be used to determine the minimum estimated duration of a project and the slack times associated with various non-critical tasks.

Thus any path whose duration equals MT is a critical path. Note that, there can be more than one critical path for a project. Tasks which fall on the critical path should receive special attention by both the project manager and the personnel assigned to perform those tasks. One way is to draw the critical paths with a double line instead of a single line. The critical path may change as the project progresses. This may happen when tasks are completed either behind or ahead of schedule.

The project parameters for different tasks for the MIS problem can be computed as follows:

1. Compute ES and EF for each task. Use the rule: `ES is equal to the largest EF the immediate predecessors`
2. Compute LS and LF for each task. Use the rule: `LF is equal to the smallest LS of the immediate successors`
3. Compute ST for each task. Use the rule: ST=LF-EF

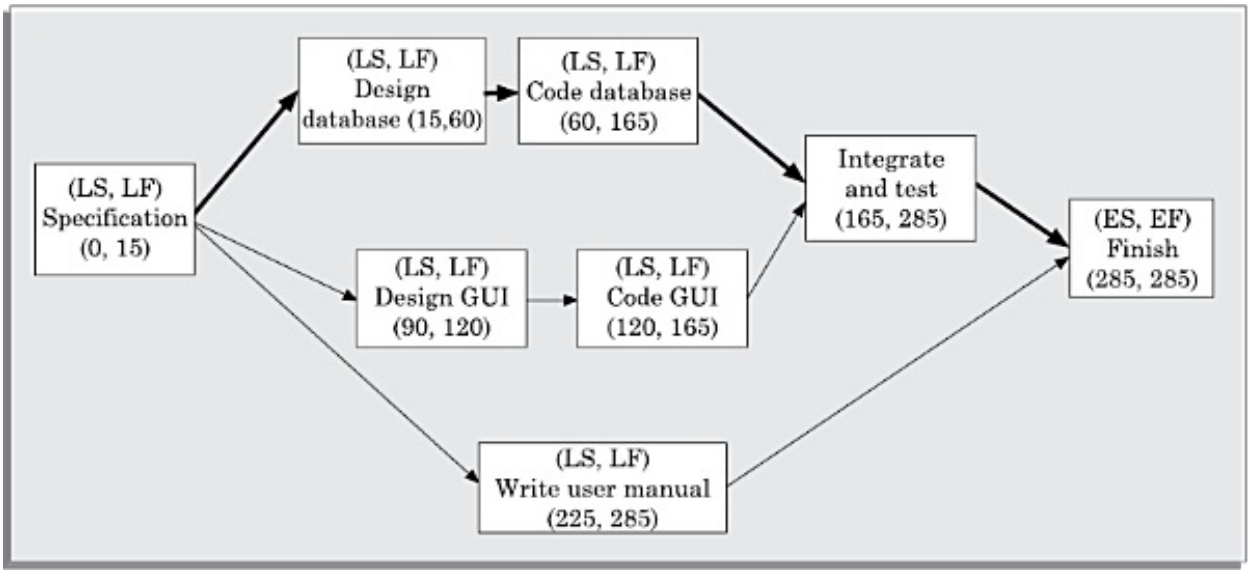

Figure 3.10: AoN of MIS problem with (LS,LF).

In Figure 3.9 and Figure 3.10, we show computation of (ES,EF) and (LS,LF) respectively. From this project parameters for different tasks for the MIS problem have been represented in Table 3.8.

Table 3.8: Project Parameters Computed From Activity Network

Task	ES	EF	LS	LF	ST
Specification	0	15	0	15	0
Design data base	15	60	15	60	0
Design GUI part	15	45	90	120	75
Code data base	60	165	60	165	0
Code GUI part	45	90	120	165	75
Integrate and test	165	285	165	285	0
Write user manual	15	75	225	285	210

The critical paths are all the paths whose duration equals MT. The critical path in Figure 3.8 is shown using a thick arrows.

3.10.4 PERT Charts

The activity durations computed using an activity network are only estimated duration. It is therefore not possible to estimate the worst case (pessimistic) and best case (optimistic) estimations using an activity diagram. Since, the actual durations might vary from the estimated durations, the utility of the activity network diagrams are limited. The CPM can be used to determine the duration of a project, but does not provide any indication of the probability of meeting that schedule.

Project evaluation and review technique (PERT) charts are a more sophisticated form of activity chart. Project managers know that there is considerable uncertainty about how much time a task would exactly take to complete. The duration assigned to tasks by the project manager are after all only estimates. Therefore, in reality the duration of an activity is a random variable with some probability distribution. In this context, PERT charts can be used to determine the probabilistic times for reaching various project mile stones, including the final mile stone. PERT charts like activity networks consist of a network of boxes and arrows. The boxes represent activities and the arrows represent task dependencies. A PERT chart represents the statistical variations in the project estimates assuming these to be normal distribution. PERT allows for some randomness in task completion times, and therefore provides the capability to determine the probability for achieving project milestones based on the probability of completing each task along the path to that milestone. Each task is annotated with three estimates:

- **Optimistic (O):** The best possible case task completion time.
- **Most likely estimate (M):** Most likely task completion time.
- **Worst case (W):** The worst possible case task completion time.

The optimistic (O) and worst case (W) estimates represent the extremities of all possible scenarios of task completion. The most likely estimate (M) is the completion time that has the highest probability. The three estimates are used to compute the expected value of the standard deviation.

It can be shown that the entire distribution lies between the interval (M – 3 □ ST) and (M + 3 □ ST), where ST is the standard deviation. From this it is possible to show that—The standard deviation for a task ST = (P – O)/6.

The mean estimated time is calculated as ET = (O + 4M + W)/6.

Since all possible completion times between the minimum and maximum duration for every task has to be considered, there can be many critical paths, depending on the various permutations of the estimates for each task. This makes critical path analysis in PERT charts very complex. A critical path in a PERT chart is shown by using thicker arrows. The PERT chart representation of the MIS problem of Figure 3.8. is shown in Figure 3.11.

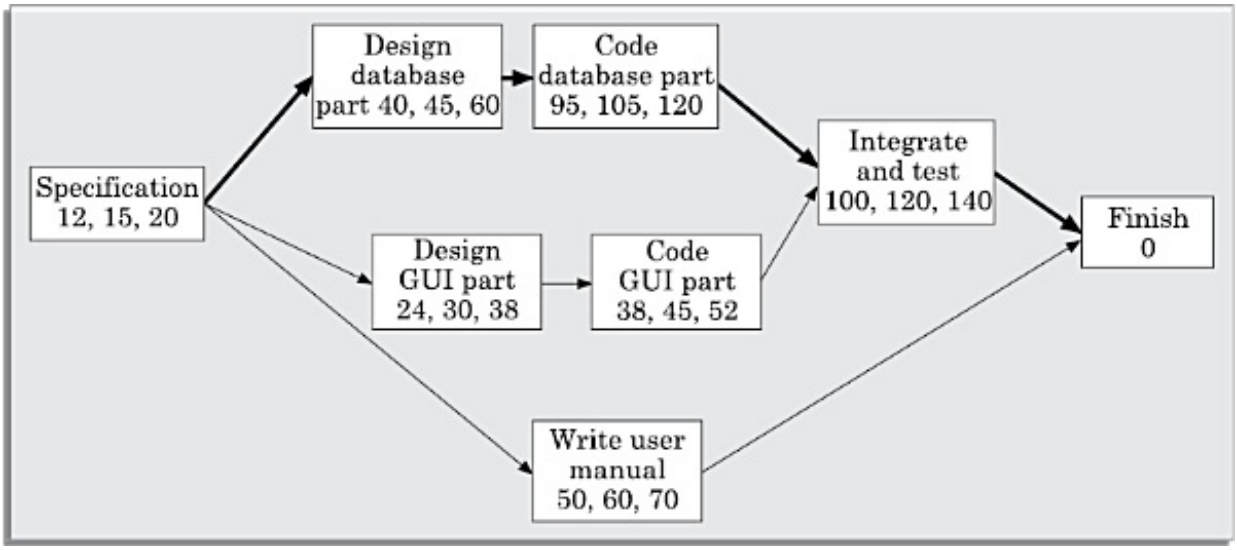

Figure 3.11: PERT chart representation of the MIS problem.

3.10.5 Gantt Charts

Gantt chart has been named after its developer Henry Gantt. A Gantt chart is a form of bar chart. The vertical axis lists all the tasks to be performed. The bars are drawn along the y-axis, one for each task. Gantt charts used in software project management are actually an enhanced version of the standard Gantt charts. In the Gantt charts used for software project management, each bar consists of a unshaded part and a shaded part. The shaded part of the bar shows the length of time each task is estimated to take. The unshaded part shows the slack time or lax time. The lax time represents the leeway or flexibility available in meeting the latest time by which a task must be finished. A Gantt chart representation for the MIS problem of Figure 3.8 is shown in Figure 3.12. Gantt charts are useful for resource planning (i.e. allocate resources to activities). The different types of resources that need to be allocated to activities include staff, hardware, and software.

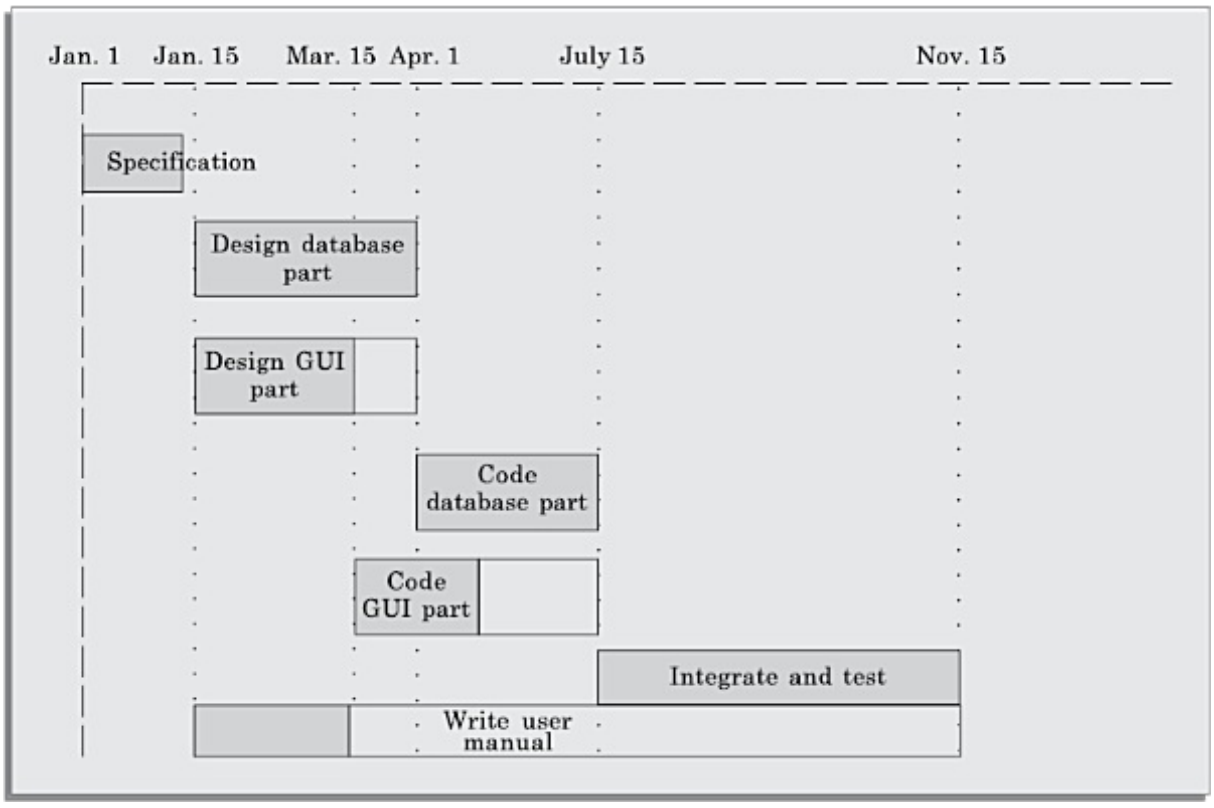

Figure 3.12: Gantt chart representation of the MIS problem.

A Gantt chart is a special type of bar chart where each bar represents an activity. The bars are drawn along a time line. The length of each bar is proportional to the duration of time planned for the corresponding activity.

Gantt chart representation of a project schedule is helpful in planning the utilisation of resources, while PERT chart is useful for monitoring the timely progress of activities. Also, it is easier to identify parallel activities in a project using a PERT chart. Project managers need to identify the parallel activities in a project for assignment to different developers.

Project monitoring and control

Once a project gets underway, the project manager monitors the project continuously to ensure that it is progressing as per plan. The project manager designates certain key events such as completion of some important activity as a **milestone**. A few examples of milestones are as following—a milestone can be the preparation and review of the SRS document, completion of the coding and unit testing, etc. Once a milestone is reached, the project manager can assume that some measurable progress has been made. If any delay in reaching a milestone is predicted, then corrective actions might have to be taken. This may entail reworking all the schedules and producing a fresh

schedule.

As already mentioned, the PERT chart is especially useful in project monitoring and control. A path in this graph is any set of consecutive nodes and edges from the starting node to the last node. A critical path in this graph is a path along which every milestone is critical to meeting the project deadline. In other words, if any delay occurs along a critical path, the entire project would get delayed. It is therefore necessary to identify all the critical paths in a schedule—adhering to the schedules of the tasks appearing on the critical paths is of prime importance to meet the delivery date. Please note that there may be more than one critical path in a schedule. The tasks along a critical path are called critical tasks. The critical tasks need to be closely monitored and corrective actions need to be initiated as soon as any delay is noticed. If necessary, a manager may switch resources from a non-critical task to a critical task so that all milestones along the critical path are met.

Several tools are available which can help you to figure out the critical paths in an unrestricted schedule, but figuring out an optimal schedule with resource limitations and with a large number of parallel tasks is a very hard problem. There are several commercial products for automating the scheduling techniques are available. Popular tools to help draw the schedule-related graphs include the MS-Project software available on personal computers.

3.11 ORGANISATION AND TEAM STRUCTURES

Usually every software development organisation handles several projects at any time. Software organisations assign different teams of developers to handle different software projects. With regard to staff organisation, there are two important issues—How is the organisation as a whole structured? And, how are the individual project teams structured? There are a few standard ways in which software organisations and teams can be structured. We discuss these in the following subsection.

3.11.1 Organisation Structure

Essentially there are three broad ways in which a software development organisation can be structured—functional format, project format, and matrix format. We discuss these three formats in the following subsection.

Functional format

In the functional format, the development staff are divided based on the specific functional group to which they belong to. This format has schematically been shown in Figure 3.13(a).

The different projects borrow developers from various functional groups for specific phases of the project and return them to the functional group upon the completion of the phase. As a result, different teams of programmers from different functional groups perform different phases of a project. For example, one team might do the requirements specification, another do the design, and so on. The partially completed product passes from one team to another as the product evolves. Therefore, the functional format requires considerable communication among the different teams and development of good quality documents because the work of one team must be clearly understood by the subsequent teams working on the project. The functional organisation therefore mandates good quality documentation to be produced after every activity.

Project format

In the project format, the development staff are divided based on the project for which they work (See Figure 3.13(b)). A set of developers is assigned to every project at the start of the project, and remain with the project till the completion of the project. Thus, the same team carries out all the life cycle activities. An advantage of the project format is that it provides job rotation. That is, every developer undertakes different life cycle activities in a project. However, it results in poor manpower utilisation, since the full project team is formed since the start of the project, and there is very little work for the team during the initial phases of the life cycle.

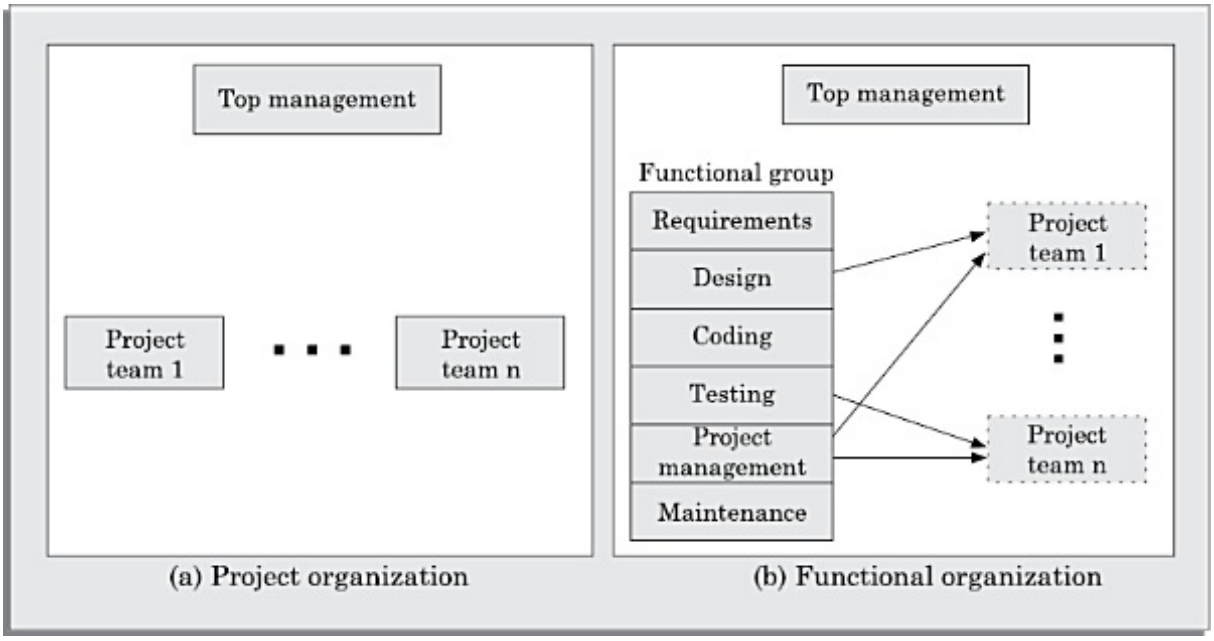

Figure 3.13: Schematic representation of the functional and project organisation.

Functional versus project formats

Even though greater communication among the team members may appear as an avoidable overhead, the functional format has many advantages. The main advantages of a functional organisation are:

- Ease of staffing
- Production of good quality documents
- Job specialisation
- Efficient handling of the problems associated with manpower turnover[3].

The functional organisation allows the developers to become specialists in particular roles, e.g. requirements analysis, design, coding, testing, maintenance, etc. They perform these roles again and again for different projects and develop deep insights to their work. It also results in more attention being paid to proper documentation at the end of a phase because of the greater need for clear communication as between teams doing different phases. The functional organisation also provides an efficient solution to the staffing problem. We have already seen in Section 3.9.2 that the staffing pattern should approximately follow the Rayleigh distribution for efficient utilisation of the personnel by minimizing their wait times. The project staffing problem is eased significantly because personnel can be brought onto a project as needed, and returned to the functional group when they are no more needed. This possibly is the most important advantage of the functional organisation.

A project organisation structure forces the manager to take in almost a

constant number of developers for the entire duration of his project. This results in developers idling in the initial phase of software development and are under tremendous pressure in the later phase of development. A further advantage of the functional organisation is that it is more effective in handling the problem of manpower turnover. This is because developers can be brought in from the functional pool when needed. Also, this organisation mandates production of good quality documents, so new developers can quickly get used to the work already done.

In spite of several important advantages of the functional organisation, it is not very popular in the software industry. This apparent paradox is not difficult to explain. We can easily identify the following three points:

- The project format provides job rotation to the team members. That is, each team member takes on the role of the designer, co der, tester, etc during the course of the project. On the other hand, considering the present skill shortage, it would be very difficult for the functional organisations to fill slots for some roles such as the maintenance, testing, and coding groups.
- Another problem with the functional organisation is that if an organisation handles projects requiring knowledge of specialized domain areas, then these domain experts cannot be brought in and out of the project for the different phases, unless the company handles a large number of such projects.
- For obvious reasons the functional format is not suitable for small organisations handling just one or two projects.

Matrix format

A matrix organisation is intended to provide the advantages of both functional and project structures. In a matrix organisation, the pool of functional specialists are assigned to different projects as needed. Thus, the deployment of the different functional specialists in different projects can be represented in a matrix (see Figure 3.14) In Figure 3.14 observe that different members of a functional specialisation are assigned to different projects. Therefore in a matrix organisation, the project manager needs to share responsibilities for the project with a number of individual functional managers.

	Project			
Functional group	#1	#2	#3	
#1	2	0	3	Functional manager 1
#2	0	5	3	Functional manager 2
#3	0	4	2	Functional manager 3
#4	1	4	0	Functional manager 4
#5	0	4	6	Functional manager 5
	Project manager 1	Project manager 2	Project manager 3	

Figure 3.14: Matrix organisation.

Matrix organisations can be characterised as weak or strong, depending upon the relative authority of the functional managers and the project managers. In a strong functional matrix, the functional managers have authority to assign workers to projects and project managers have to accept the assigned personnel. In a weak matrix, the project manager controls the project budget, can reject workers from functional groups, or even decide to hire outside workers.

Two important problems that a matrix organisation often suffers from are:

- Conflict between functional manager and project managers over allocation of workers.
- Frequent shifting of workers in a firefighting mode as crises occur in different projects.

3.11.2 Team Structure

Team structure addresses organisation of the individual project teams. Let us examine the possible ways in which the individual project teams are organised. In this text, we shall consider only three formal team structures—democratic, chief programmer, and the mixed control team organisations, although several other variations to these structures are possible. Projects of specific complexities and sizes often require specific team structures for efficient working.

Chief programmer team

In this team organisation, a senior engineer provides the technical leadership and is designated the chief programmer. The chief programmer partitions the task into many smaller tasks and assigns them to the team members. He also

verifies and integrates the products developed by different team members. The structure of the chief programmer team is shown in Figure 3.15. The chief programmer provides an authority, and this structure is arguably more efficient than the democratic team for well-understood problems. However, the chief programmer team leads to lower team morale, since the team members work under the constant supervision of the chief programmer. This also inhibits their original thinking. The chief programmer team is subject to single point failure since too much responsibility and authority is assigned to the chief programmer. That is, a project might suffer severely, if the chief programmer either leaves the organisation or becomes unavailable for some other reasons.

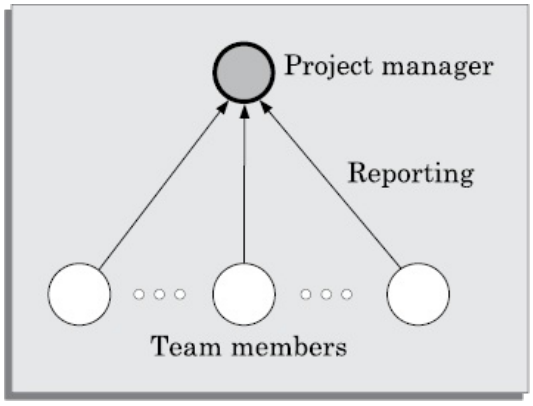

Figure 3.15: Chief programmer team structure.

The chief programmer team is probably the most efficient way of completing simple and small projects since the chief programmer can quickly work out a satisfactory design and ask the programmers to code different modules of his design solution.

Let us now try to understand the types of projects for which the chief programmer team organisation would be suitable. Suppose an organisation has successfully completed many simple MIS projects. Then, for a similar MIS project, chief programmer team structure can be adopted. The chief programmer team structure works well when the task is within the intellectual grasp of a single individual. However, even for simple and well-understood problems, an organisation must be selective in adopting the chief programmer structure. The chief programmer team structure should not be used unless the importance of early completion outweighs other factors such as team morale, personal developments, etc.

Democratic team

The democratic team structure, as the name implies, does not enforce any formal team hierarchy (see Figure 3.16). Typically, a manager provides the

administrative leadership. At different times, different members of the group provide technical leadership.

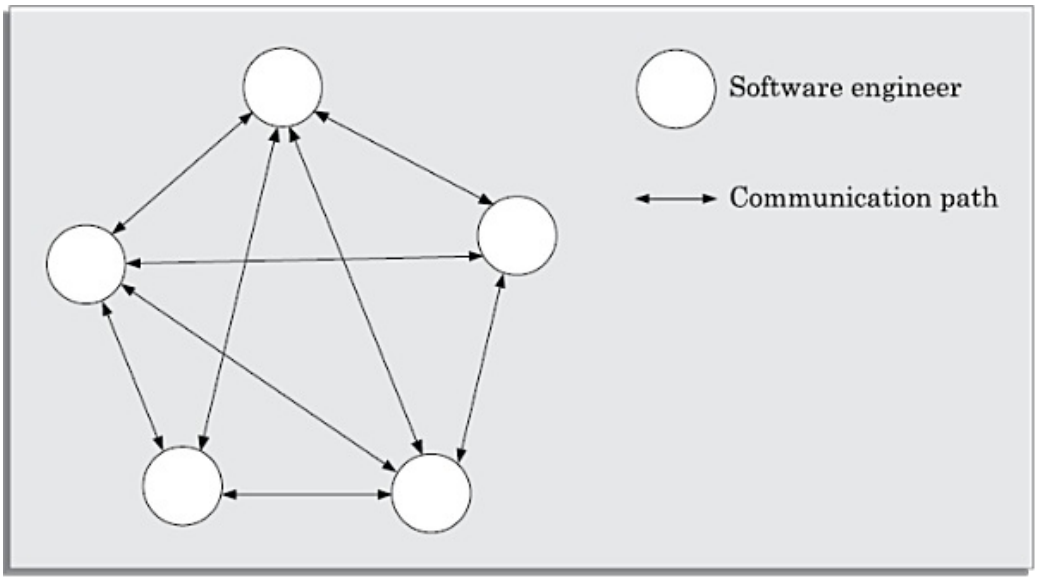

Figure 3.16: Democratic team structure.

In a democratic organisation, the team members have higher morale and job satisfaction. Consequently, it suffers from less manpower turnover. Though the democratic teams are less productive compared to the chief programmer team, the democratic team structure is appropriate for less understood problems, since a group of developers can invent better solutions than a single individual as in a chief programmer team. A democratic team structure is suitable for research-oriented projects requiring less than five or six developers. For large sized projects, a pure democratic organisation tends to become chaotic. The democratic team organisation encourages egoless programming as programmers can share and review each other's work. To appreciate the concept of egoless programming, we need to understand the concept of ego from a psychological perspective.

Most of you might have heard about temperamental artists who take much pride in whatever they create. Ordinarily, the human psychology makes an individual take pride in everything he creates using original thinking. Software development requires original thinking too, although of a different type. The human psychology makes one emotionally involved with his creation and hinders him from objective examination of his creations. Just like temperamental artists, programmers find it extremely difficult to locate bugs in their own programs or flaws in their own design. Therefore, the best way to find problems in a design or code is to have someone review it. Often, having to explain one's program to someone else gives a person enough

objectivity to find out what might have gone wrong. This observation is the basic idea behind code walk throughs to be discussed in Chapter 10. An application of this, is to encourage a democratic teams to think that the design, code, and other deliverables to belong to the entire group. This is called egoless programming because it tries to avoid having programmers invest much ego in the development activity they do in a democratic set up. However, a democratic team structure has one disadvantage—the team members may waste a lot time arguing about trivial points due to the lack of any authority in the team to resolve the debates.

Mixed control team organisation

The mixed control team organisation, as the name implies, draws upon the ideas from both the democratic organisation and the chief-programmer organisation. The mixed control team organisation is shown pictorially in Figure 3.17. This team organisation incorporates both hierarchical reporting and democratic set up. In Figure 3.17, the communication paths are shown as dashed lines and the reporting structure is shown using solid arrows. The mixed control team organisation is suitable for large team sizes. The democratic arrangement at the senior developers level is used to decompose the problem into small parts. Each democratic setup at the programmer level attempts solution to a single part. Thus, this team organisation is eminently suited to handle large and complex programs. This team structure is extremely popular and is being used in many software development companies.

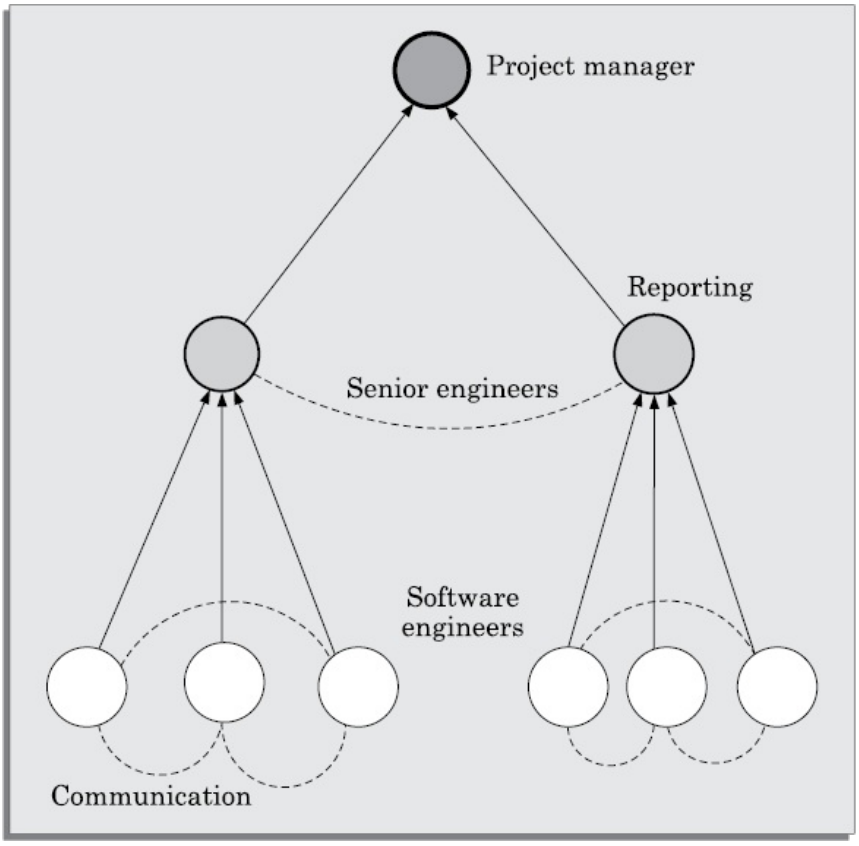

Figure 3.17: Mixed team structure.

3.12 STAFFING

Software project managers usually take the responsibility of choosing their team. Therefore, they need to identify good software developers for the success of the project. A common misconception held by managers as evidenced in their staffing, planning and scheduling practices, is the assumption that one software engineer is as productive as another. However, experiments have revealed that there exists a large variability of productivity between the worst and the best software developers in a scale of 1 to 30. In fact, the worst developers may sometimes even reduce the overall productivity of the team, and thus in effect exhibit negative productivity. Therefore, choosing good software developers is crucial to the success of a project.

Who is a good software engineer?

In the past, several studies concerning the traits of a good software engineer have been carried out. All these studies roughly agree on the following attributes that good software developers should possess:

- Exposure to systematic techniques, i.e. familiarity with software

engineering principles.
- Good technical knowledge of the project areas (Domain knowledge)
- Good programming abilities.
- Good communication skills. These skills comprise of oral, written, and interpersonal skills.
- High motivation.
- Sound knowledge of fundamentals of computer science
- Intelligence.
- Ability to work in a team.
- Discipline, etc.

Studies show that these attributes vary as much as 1:30 for poor and bright candidates. An experiment conducted by Sackman [1968] shows that the ratio of coding hour for the worst to the best programmers is 25:1, and the ratio of debugging hours is 28:1. Also, the ability of a software engineer to arrive at the design of the software from a problem description varies greatly with respect to the parameters of quality and time.

Technical knowledge in the area of the project (domain knowledge) is an important factor determining the productivity of an individual for a particular project, and the quality of the product that he develops. A programmer having a thorough knowledge of database applications (e.g. MIS) may turn out to be a poor data communication developer. Lack of familiarity with the application areas can result in low productivity and poor quality of the product.

Since software development is a group activity, it is vital for a software developer to possess three main kinds of communication skills—Oral, Written, and Interpersonal. A software developer not only needs to effectively communicate with his teammates (e.g. reviews, walk throughs, and other team communications) but may also have to communicate with the customer to gather product requirements. Poor interpersonal skills hamper these vital activities and often show up as poor quality of the product and low productivity. Software developers are also required at times to make presentations to the managers and to the customers. This requires a different kind of communication skill (oral communication skill). A software developer is also expected to document his work (design, code, test, etc.) as well as write the users' manual, training manual, installation manual, maintenance manual, etc. This requires good written communication skill.

Motivation level of a software developer is another crucial factor contributing to his work quality and productivity. Even though no systematic studies have been reported in this regard, it is generally agreed that even bright developers may turn out to be poor performers when they lack motivation. An average developer who can work with a single mind track can outperform other developers. But motivation is a complex phenomenon requiring careful control. For majority of software developers, higher incentives and better working conditions have only limited affect on their motivation levels. Motivation is to a great extent determined by personal traits, family and social backgrounds, etc.

3.13 RISK MANAGEMENT

Every project is susceptible to a large number of risks. Without effective management of the risks, even the most meticulously planned project may go hay ware.

> A risk is any anticipated unfavourable event or circumstance that can occur while a project is underway.

We need to distinguish between a risk which is a problem that might occur from the problems currently being faced by a project. If a risk becomes real, the anticipated problem becomes a reality and is no more a risk. If a risk becomes real, it can adversely affect the project and hamper the successful and timely completion of the project. Therefore, it is necessary for the project manager to anticipate and identify different risks that a project is susceptible to, so that contingency plans can be prepared beforehand to contain each risk. In this context, risk management aims at reducing the chances of a risk becoming real as well as reducing the impact of a risks that becomes real. Risk management consists of three essential activities—risk identification, risk assessment, and risk mitigation. We discuss these three activities in the following subsections.

3.13.1 Risk Identification

The project manager needs to anticipate the risks in a project as early as possible. As soon as a risk is identified, effective risk management plans are made, so that the possible impacts of the risks is minimised. So, early risk identification is important. Risk identification is somewhat similar to the project manager listing down his nightmares. For example, project manager might be worried whether the vendors whom

you have asked to develop certain modules might not complete their work in time, whether they would turn in poor quality work, whether some of your key personnel might leave the organisation, etc. All such risks that are likely to affect a project must be identified and listed.

A project can be subject to a large variety of risks. In order to be able to systematically identify the important risks which might affect a project, it is necessary to categorise risks into different classes. The project manager can then examine which risks from each class are relevant to the project. There are three main categories of risks which can affect a software project: project risks, technical risks, and business risks. We discuss these risks in the following.

Project risks: Project risks concern various forms of budgetary, schedule, personnel, resource, and customer-related problems. An important project risk is schedule slippage. Since, software is intangible, it is very difficult to monitor and control a software project. It is very difficult to control something which cannot be seen. For any manufacturing project, such as manufacturing of cars, the project manager can see the product taking shape. He can for instance, see that the engine is fitted, after that the doors are fitted, the car is getting painted, etc. Thus he can accurately assess the progress of the work and control it, if he finds any activity is progressing at a slower rate than what was planned. The invisibility of the product being developed is an important reason why many software projects suffer from the risk of schedule slippage.

Technical risks: Technical risks concern potential design, implementation, interfacing, testing, and maintenance problems. Technical risks also include ambiguous specification, incomplete specification, changing specification, technical uncertainty, and technical obsolescence. Most technical risks occur due the development team's insufficient knowledge about the product.

Business risks: This type of risks includes the risk of building an excellent product that no one wants, losing budgetary commitments, etc.

Classification of risks in a project

Example 3.12 Let us consider a satellite based mobile communication product discussed in Case Study 2.2 of Section 2.5. The project manager can identify several risks in this project. Let us classify them appropriately.

- What if the project cost escalates and overshoots what was

estimated?: **Project risk.**

- What if the mobile phones that are developed become too bulky in size to conveniently carry?: **Business risk.**
- What if it is later found out that the level of radiation coming from the phones is harmful to human being?: **Business risk.**
- What if call hand-off between satellites becomes too difficult to implement?: **Technical risk.**

In order to be able to successfully foresee and identify different risks that might affect a software project, it is a good idea to have a company disaster list. This list would contain all the bad events that have happened to software projects of the company over the years including events that can be laid at the customer's doors. This list can be read by the project mangers in order to be aware of some of the risks that a project might be susceptible to. Such a disaster list has been found to help in performing better risk analysis.

3.13.2 Risk Assessment

The objective of risk assessment is to rank the risks in terms of their damage causing potential. For risk assessment, first each risk should be rated in two ways:

- The likelihood of a risk becoming real (r).
- The consequence of the problems associated with that risk (s).

Based on these two factors, the priority of each risk can be computed as follows:

$$p = r \times s$$

where, p is the priority with which the risk must be handled, r is the probability of the risk becoming real, and s is the severity of damage caused due to the risk becoming real. If all identified risks are prioritised, then the most likely and damaging risks can be handled first and more comprehensive risk abatement procedures can be designed for those risks.

3.13.3 Risk Mitigation

After all the identified risks of a project have been assessed, plans are made to contain the most damaging and the most likely risks first. Different types of risks require different containment procedures. In

fact, most risks require considerable ingenuity on the part of the project manager in tackling the risks.

There are three main strategies for risk containment:

Avoid the risk: Risks can be avoided in several ways. Risks often arise due to project constraints and can be avoided by suitably modifying the constraints. The different categories of constraints that usually give rise to risks are:

Process-related risk: These risks arise due to aggressive work schedule, budget, and resource utilisation.

Product-related risks: These risks arise due to commitment to challenging product features (e.g. response time of one second, etc.), quality, reliability etc.

Technology-related risks: These risks arise due to commitment to use certain technology (e.g., satellite communication).

A few examples of risk avoidance can be the following: Discussing with the customer to change the requirements to reduce the scope of the work, giving incentives to the developers to avoid the risk of manpower turnover, etc.

Transfer the risk: This strategy involves getting the risky components developed by a third party, buying insurance cover, etc.

Risk reduction: This involves planning ways to contain the damage due to a risk. For example, if there is risk that some key personnel might leave, new recruitment may be planned. The most important risk reduction techniques for technical risks is to build a prototype that tries out the technology that you are trying to use. For example, if you are using a compiler for recognising user commands, you would have to construct a compiler for a small and very primitive command language first.

There can be several strategies to cope up with a risk. To choose the most appropriate strategy for handling a risk, the project manager must consider the cost of handling the risk and the corresponding reduction of risk. For this we may compute the risk leverage of the different risks. Risk leverage is the difference in risk exposure divided by the cost of reducing the risk. More formally,

$$\text{risk leverage} = \frac{\text{risk exposure before reduction} - \text{risk exposure after reduction}}{\text{cost of reduction}}$$

Even though we identified three broad ways to handle any risk, effective risk handling cannot be achieved by mechanically following a set procedure,

but requires a lot of ingenuity on the part of the project manager. As an example, let us consider the options available to contain an important type of risk that occurs in many software projects—that of schedule slippage.

An example of handling schedule slippage risk

Risks relating to schedule slippage arise primarily due to the intangible nature of software. For a project such as building a house, the progress can easily be seen and assessed by the project manager. If he finds that the project is lagging behind, then corrective actions can be initiated. Considering that software development per se is invisible, the first step in managing the risks of schedule slippage, is to increase the visibility of the software product. Visibility of a software product can be increased by producing relevant documents during the development process and getting these documents reviewed by an appropriate team. Milestones should be placed at regular intervals to provide a manager with regular indication of progress. Completion of a phase of the development process being followed need not be the only milestones. Every phase can be broken down to reasonable-sized tasks and milestones can be associated with these tasks. A milestone is reached, once documentation produced as part of a software engineering task is produced and gets successfully reviewed. Milestones need not be placed for every activity. An approximate rule of thumb is to set a milestone every 10 to 15 days. If milestones are placed too close each other than the overheads in managing the milestones would be too much.

3.14 SOFTWARE CONFIGURATION MANAGEMENT

The results (also called as the deliverables) of a large software development effort typically consist of a large number of objects, e.g., source code, design document, SRS document, test document, user's manual, etc. These objects are usually referred to and modified by a number of software developers through out the life cycle of the software. The state of each deliverable object changes as development progresses and also as bugs are detected and fixed.

> The configuration of the software is the state of all project deliverables at any point of time; and software configuration management deals with effectively tracking and controlling the configuration of a software during its life cycle.

As a software is changed, new revisions and versions get created. Before we discuss configuration management, we must be clear about the distinction

between a version and a revision of a software product.

Software revision versus version

A new version of a software is created when there is significant change in functionality, technology, or the hardware it runs on, etc. On the other hand, a new release is created if there is only a bug fix, minor enhancements to the functionality, usability, etc. Even the initial delivery might consist of several versions and more versions might be added later on.

For example, one version of a mathematical computation package might run on Unix-based machines, another on Microsoft Windows and so on. As a software is released and used by the customer, errors are discovered that need correction. Enhancements to the functionalities of the software may also be needed. A new release of software is an improved system intended to replace an old one. Often systems are described as version m, release n; or simply mn. Formally, a history relation is version of can be defined between objects. This relation can be split into two subrelations is revision of and is variant of. In the following subsections, we first discuss the necessity of configuration management and subsequently we discuss the configuration management activities and tools.

3.14.1 Necessity of Software Configuration Management

There are several reasons for putting an object under configuration management. The following are some of the important problems that can crop up, if configuration management is not used: every software developer has a personal copy of an object (e.g. source code). When a developer makes changes to his local copy, he is expected to intimate the changes that he has made to other developers, so that the necessary changes in interfaces could be uniformly carried out across all modules. However, not only would it eat up valuable time of the developers, but many times a developer might make changes to the interfaces in his local copies and forgets to intimate the teammates about the changes. This makes the different copies of the object inconsistent. Finally, when the different modules are integrated, it does not work. Therefore, when several team members work on developing an application, it is necessary for them to work on a single copy of the application, otherwise inconsistencies may arise.

Problems associated with concurrent access: Possibly the most

important reason for configuration management is to control the access to the different deliverable objects. Unless strict discipline is enforced regarding updation and storage of different objects, several problems can appear. Assume that only a single copy of a program module is maintained, and several developer are working on it. Two developers may simultaneously carry out changes to different functions of the same module, and while saving overwrite each other. Similar problems can occur for any other deliverable object.

Providing a stable development environment: When a project work is underway, the team members need a stable environment to make progress. Suppose one developer is trying to integrate module A, with the modules B and C; since if developer of module C keeps changing C; this can be especially frustrating if a change to module C forces recompilation of the module. When an effective configuration management is in place, the manager freezes the objects to form a baseline.

A baseline is the status of all the objects under configuration control. When any of the objects under configuration control is changed, a new baseline gets formed.

When any team member needs to change any of the objects under configuration control, he is provided with a copy of the baseline item. The requester makes changes to his private copy. Only after the requester is through with all modifications to his private copy, the configuration is updated and a new baseline gets formed instantly. This establishes a baseline for others to use and depend on. Also, baselines may be archived periodically (archiving means copying to a safe place such as a remote storage), so that the last baseline can be recovered when there is a disaster.

System accounting and maintaining status information: System accounting denotes keeping track of who made a particular change to an object and when the change was made.

Handling variants: Existence of variants of a software product causes some peculiar problems. Suppose you have several variants of the same module, and find that a bug exists in one of them. Then, it has to be fixed in all versions and revisions. To do it efficiently, you should not have to fix it in each and every version and revision of the software separately. Making a change to one program should be reflected appropriately in all relevant versions and revisions.

3.14.2 Configuration Management Activities

Configuration management is carried out through two principal activities:

Configuration identification: I t involves deciding which parts of the system should be kept track of.

Configuration control: It ensures that changes to a system happen smoothly. Normally, a project manager performs the configuration management activity by using a configuration management tool. In addition, a configuration management tool helps to keep track of various deliverable objects, so that the project manager can quickly and unambiguously determine the current state of the project. The configuration management tool enables the developer to change various components in a controlled manner.

In the following subsections, we provide an overview of the two configuration management activities.

Configuration identification

Project managers normally classify the objects associated with a software development into three main categories—controlled, precontrolled, and uncontrolled. Controlled objects are those that are already under configuration control. The team members must follow some formal procedures to change them. Precontrolled objects are not yet under configuration control, but will eventually be under configuration control. Uncontrolled objects are not subject to configuration control. Controllable objects include both controlled and precontrolled objects. Typical controllable objects include:

- Requirements specification document
- Design documents
- Tools used to build the system, such as compilers, linkers, lexical analysers, parsers, etc.
- Source code for each module
- Test cases
- Problem reports

Configuration management plan is written during the project planning phase. It lists all controlled objects. The managers who develop the plan must strike a balance between controlling too much, and controlling too little. If too much is controlled, overheads due to configuration management

increase to unreasonably high levels. On the other hand, controlling too little might lead to confusion and inconsistency when something changes.

Configuration control

Configuration control is the process of managing changes to controlled objects. The configuration control part of a configuration management system that most directly affects the day-to-day operations of developers.

> Configuration control allows only authorised changes to the controlled objects to occur and prevents unauthorised changes.

In order to change a controlled object such as a module, a developer can get a private copy of the module by a reserve operation (see Figure 3.18). Configuration management tools allow only one person to reserve a module at any time. Once an object is reserved, it does not allow any one else to reserve this module until the reserved module is restored. Thus, by preventing more than one developer to simultaneously reserve a module, the problems associated with concurrent access are solved.

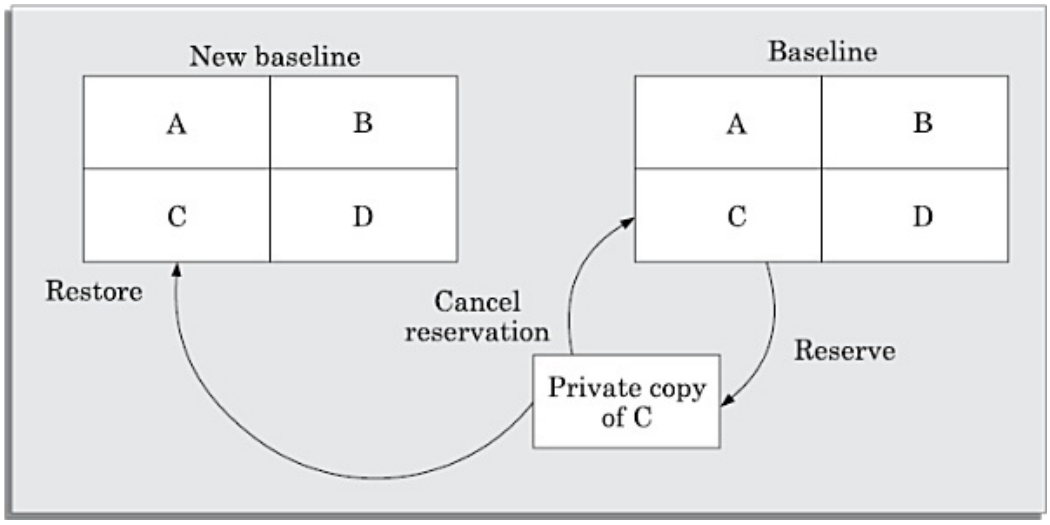

Figure 3.18: Reserve and restore operation in configuration control.

Let us see how an object under configuration control can be changed. The developer needing to change a module first makes a reserve request. After the reserve command successfully executes, a private copy of the module is created in his local directory. Then, he carries out all necessary changes on his private copy. Once has satisfactorily completes all necessary changes, the changes need to be restored in configuration management repository. However, restoring the changed module to the system configuration requires the permission of a change control board (CCB). The CCB is usually constituted from among the development team members. For every change

that needs to be carried out, the CCB reviews the changes made to the controlled object and certifies several things about the change:

1. Change is well-motivated.
2. Developer has considered and documented the effects of the change.
3. Changes interact well with the changes made by other developers.
4. Appropriate people (CCB) have validated the change, e.g., someone has tested the changed code, and has verified that the change is consistent with the requirement.

The change control board (CCB) sounds like a group of people. However, except for very large projects, the functions of the change control board are normally discharged by the project manager himself or some senior member of the development team. Once the CCB reviews the changes to the module, the project manager updates the old baseline through a restore operation (see Figure 3.18). A configuration control tool does not allow a developer to replace an object he has reserved with his local copy unless he gets an authorisation from the CCB. By constraining the developers' ability to replace reserved objects, a stable environment is achieved. Since a configuration management tool allows only one developer to work on one module at any one time, problem of accidental overwriting is eliminated. Also, since only the manager can update the baseline after the CCB approval, unintentional changes to the configuration items are eliminated.

Source code control system (SCCS) and RCS

SCCS and RCS are two popular configuration management tools available on most Unix systems. SCCS or RCS can be used for controlling and managing different versions of text files. SCCS and RCS do not handle binary files (i.e., executable files, documents, files containing diagrams, etc.) SCCS and RCS provide an efficient way of storing versions that minimises the amount of occupied disk space. Suppose, a module MOD is present in 3 versions—MOD1.1, MOD1.2 and MOD1.3. Then, SCCS and RCS stores the original module MOD1.1 together with changes needed to transform MOD1.1 into MOD1.2 and MOD1.2 to MOD1.3. The changes needed to transform each baselined file to the next version are stored and are called deltas. The main reason behind storing the deltas rather than storing the full revision files is to save disk space.

The change control facilities provided by SCCS and RCS include the ability

to incorporate restrictions on the set of individuals who can create new versions, and facilities for checking components in and out (i.e., reserve and restore operations). Individual developers check out components and modify them. After they have made all necessary changes to a module and after the changes have been reviewed, they check in the changed module into SCCS or RCS. Revisions are denoted by numbers in ascending order, e.g., 1.1, 1.2, 1.3, etc. It is also possible to create variants or revisions of a component by creating a fork in the development history.

3.15 MISCELLANEOUS PLANS

Besides cost estimation, scheduling, and staffing plans, project managers plan several other things during the project planning stage. An important task at this stage is the selection of a suitable development process model. We have already discussed in Chapter 2 that different problems require either adopting an entirely different process model or suitably tailoring the standard process model adopted by a company. For example, a routine development work may require a waterfall model to be followed, and an ambitious and technically challenging project may require an evolutionary or prototyping development model to be adopted. Also, depending upon the type of the project, some life cycle phases may be omitted, modified, or new phases may be added to a selected life cycle model. For example, for a software maintenance project, the design phase may be modified to a design modification stage. Thus during the project planning stage, the project manager may have to select a suitable process model and do any required tailoring.

SUMMARY

- In this chapter, we examined the chief responsibilities of a software project manager. We mentioned that various responsibilities of a project manager can be classified into two broad classes:

– Project planning, and
– Project monitoring and control.

- Project planning activities are undertaken before any development activity starts. We discussed the important project planning activities—estimation, scheduling, and staffing.

- We also discussed about the other plans that project mangers must do during the project planning stage, such as risk analysis, configuration plan, and process tailoring.
- Project monitoring and control activities are undertaken after the development work starts.
- We emphasised that although some systematic techniques are available to the project managers to do project planning; for project monitoring and control, experience and subjective judgement are very important.
- Software configuration is the state of the deliverable items at any point in time. We discussed that without proper configuration management, a project would suffer from many types of problems. For this reason, software configuration management is an important and mandatory requirement in almost all software quality assurance principles.

EXERCISES

1. Choose the correct option:
   (a) Effort is measured using which one of the following units:
   (i) persons
   (ii) person-months
   (iii) months
   (iv) Rupees
   (b) COCOMO estimation model can be used to estimate which one of the following:
   (i) LOC
   (ii) Effort
   (iii) Function points
   (iv) Defect density
   (c) What is the correct order in which a software project manager estimates various project parameters while using COCOMO:
   (i) Cost, effort, duration, size
   (ii) Cost, duration, effort, size
   (iii) Size, effort, duration, cost
   (iv) Size, cost, effort, duration
   (d) Which one of the following is NOT a factor for "Lines of code" being considered as a poor size metric:
   (i) It is programming language dependent. (ii) It penalises efficient and compact coding.

(iii) It is programmer dependent.
(iv) It is dependent on the complexity of the requirements.

(e) Which one of the following project parameters is usually the first to be estimated by a project manager:
(i) Cost
(ii) Effort
(iii) Size
(iv) Duration

(f) Which one of the following charts is the most useful to decompose the project activities into smaller tasks that can be more meaningfully managed:
(i) PERT chart
(ii) GANTT chart
(iii) Task network representation
(iv) Work breakdown structure

(g) Which one of the following is an example of a multivariable cost estimation model?
(i) Basic COCOMO
(ii) Intermediate COCOMO
(iii) Complete COCOMO
(iv) Delphi technique

(h) If a software product of size S takes m months to develop, then according to the COCOMO estimation model, how long (in months) will it take to develop a product of size 2 × S?
(i) Greater than 2 × m months
(ii) Greater than 3 × m months
(iii) Less than 2 × m months
(iv) Greater than 4 × m months

(i) Which of the following statements is true of the COCOMO model.
(i) Cost is the most fundamental attribute of a software product, based on which the project size and duration are measured.
(ii) Size is the most fundamental attribute of a software product, based on which the project cost and duration are measured.
(iii) Effort is the most fundamental attribute of a software product, based on which the project size and cost are measured.
(iv) Duration is the most fundamental attribute of a software product, based on which the project size and effort are measured.

(j) For a certain software development project, an effort estimation of

100 person- months was arrived by using COCOMO model. This implies that the project needs to be completed by:

(i) Employing 100 persons for 1 month
(ii) Employing 1 person for 100 months
(iii) Employing 10 persons for 10 months
(iv) The number of persons employed over different project phases would correspond to Raleigh distribution

(k) Which one of the following most closely describes configuration management in software engineering?

(i) Management of the configuration parameter settings in the software.
(ii) Management of objects that control the system configuration parameter settings.
(iii) Management of the states of various project deliverables.
(iv) Configuration of the management activities depending of the type of the projects.

(l) How is an application's "version" different from its "release"?

(i) A release is a small change to an earlier release.
(ii) A version is a small change made to an earlier release.
(iii) A release is essentially the same as a version.
(iv) A release is the one made available to customers whereas versions are for internal use.

(m) If a project is already delayed, then adding manpower to complete it at the earliest would be:

(i) Always counter productive
(ii) Can help to a very limited extent
(iii) Most effective way to tackle the situation
(iv) Can cause project completion in the shortest time

2. Write five major responsibilities of a software project manager.
3. Identify the factors that make software projects much more difficult to manage, compared to many other types of projects such as a project to lay out a 100 km concrete road on an existing non-concrete road.
4. At which point in the software development life cycle (SDLC), does the project management activities start? When do these end? Identify the important project management activities.
5. What is meant by the 'size' of a software project? Why does a project manager need to estimate the size of the project? How is the size estimated?

6. What do you understand by sliding window planning? Explain using a few examples the types of projects for which this form of planning is especially suitable. Is sliding window planning appropriate for small projects? What are its advantages over conventional planning?
7. What do you understand by product visibility in the context of software development?
   Why is it important to improve product visibility during software development? How can product visibility be improved.
8. What are the different categories of software development projects according to the COCOMO estimation model? Give an example of software product development projects belonging to each of these categories.
9. Briefly explain the main differences between the original COCOMO estimation model and the COCOMO 2 estimation model.
10. What do you mean by project size? What are the popular metrics to measure project size? How can the size of a project be estimated during the project planning stage?
11. Briefly explain project size estimation using Delphi and expert judgement techniques.
    Compare the advantages and disadvantages of the following two project size estimation techniques—expert judgement and Delphi technique.
12. Why is it difficult to accurately estimate the effort required for completing a project?
    Briefly explain the different effort estimation methods that are available. Which one would be the most advisable to use and why?
13. For the same number of lines of code and the same development team size, rank the following software projects in order of their estimated development time. Briefly mention the reasoning behind your answer.
    (a) A text editor
    (b) An employee pay roll system
    (c) An operating system for a new computer
14. As the the manager of a software project to develop a product for business application, if you estimate the effort required for completion of the project to be 50 person-months, can you complete the project by employing 50 developers for a period of one month? Justify your answer.
15. Briefly explain the COCOMO 2 model. In what aspects is it an improvement over the original COCOMO model.
16. State whether the following statements are **TRUE** o r **FALSE**. Give

reasons for your answer.

(a) As a project manager it would be worthwhile on your part to reduce the project duration by half provided the customer agrees to pay for the increased manpower requirements.

(b) Software organisations achieve more efficient manpower utilisation by adopting a project-based organisation structure as compared to a function-based organisation.

(c) For the development of the same product, the larger is the size of a software development team, the faster is the product development. (for simplicity, assume that all developer are equally proficient and have exactly similar experience).

(d) The number of development personnel required for any software development project can be determined by dividing the total (estimated) effort by the total (estimated) duration of the project.

(e) The democratic team organisation is very well suited to handle complex and challenging projects.

(f) It is possible to carry out the configuration management for a software project without using an automated tool.

(g) According to the COCOMO model, cost is the most fundamental attribute of a software product, based on which size and effort are estimated.

(h) Size of a project, as used in COCOMO is the size of the final executable code in bytes.

(i) Delphi estimation technique usually gives a more accurate estimation of project size compared to the expert judgement technique.

(j) A democratic team structure is the most suitable compared to other types of team structures. for developing a very large software product.

(k) When a task along a critical path is completed in less time than originally estimated, it should result in faster completion of the overall project.

(l) A task completing after its latest finish (LF) time would show up as a delay in the completion of the project by corresponding time.

(m) Project managers normally use GANTT charts for doing resource allocation, whereas PERT charts are used for monitoring and controlling the progress of the project.

17. What is a milestone in software development? Why is it considered helpful to have milestones in software development?

18. What is the order in which the following are estimated in the COCOMO

estimation technique: cost, effort, duration, size? Represent the precedence ordering among these activities using a task network diagram.

19. Explain why the development time of a software product of given size remains almost the same, regardless of whether it is organic, semidetached, or embedded type.
20. Explain why according to the COCOMO model, when the size of a software is increased by two times, the time to develop the product usually increases by less than two times.
21. Suppose you have estimated the nominal development time of a moderate-sized software product to be 5 months. You have also estimated that it will cost Rs. 50,000 to develop the software product. Now, the customer comes and tells you that he wants you to accelerate the delivery time by 10 per cent. How much additional cost would you charge the customer for this accelerated delivery? Irrespective of whether you take less time or more time to develop the product, you are essentially developing the same product. Why then does the effort depend on the duration over which you develop the product?
22. Explain how Putnam's model can be used to compute the change in project cost with project duration. What are the main disadvantages of using the Putnam's model to compute the additional costs incurred due to schedule compression? How can you overcome them?
23. Suppose you are developing a software product of organic type. You have estimated the size of the product to be about 100,000 lines of code. Compute the nominal effort and the development time.
24. For the following C program estimate the Halstead's length and volume measures. Compare Halstead's length and volume measures of size with the LOC measure.

```
/* Program to calculate GCD of two numbers */
  int compute_gcd(int x, int y)
  {
    while (x != y)
      if (x>y) then x=x-y;
      else y=y-x;
    return x;
  }
```

25. What does Halstead's volume metric represent conceptually? How according to Halstead is the effort dependent on program volume?

26. What are the relative advantages of using either the LOC or the function point metric to measure the size of a software product for software project planning?
27. List the important shortcomings of LOC for use as a software size metric for carrying out project estimations.
28. Explain why adding more man power to an already late project makes it later.
29. What do you understand by work breakdown in project management? Why is work breakdown important to effective project management? How is work breakdown achieved? What problems might occur is tasks are either broken down into too fine a granularity or tasks are broken into too coarse granularity?
30. The following table indicates the various tasks involved in completing a software project, the corresponding activities, and the estimated effort for each task in person-months.

Notation	Activity	Effort in person-months
T_1	Requirements specification	1
T_2	Design	2
T_3	Code actuator interface module	2
T_4	Code sensor interface module	5
T_5	Code user interface part	3
T_6	Code control processing part	1
T_7	Integrate and test	6
T_8	Write user manual	3

The precedence relation $T_i \leq \{T_j, T_k\}$ implies that the task T_i must complete before either task T_j or T_k can start. The following precedence relation is known to hold among different tasks: $T_1 \leq T_2 \leq \{T_3, T_4, T_5, T_6\} \leq T_7$.

(a) Draw the Activity network representation of the tasks.
(b) Determine ES, EF and LS, LF for every task.
(c) Develop the Gantt chart representations for the project.

31. Suppose you are the project manager of a software project requiring the following activities.

Activity No.	Activity Name	Duration (weeks)	Immediate Predecessor
1.	Obtain requirements	4	-

2.	Analyse operations	4	-
3.	Define subsystems	2	1
4.	Develop database	4	1
5.	Make decision analysis	3	2
6.	Identify constraints	2	5
7.	Build module 1	8	3,4,6
8	Build module 2	12	3,4,6
9.	Build module 3	18	3,4,6
10.	Write report	10	6
11.	Integration and test	8	7,8,9
12.	Implementation	2	10,11

(a) Draw the Activity Network representation of the project.
(b) Determine ES, EF and LS, LF for every task.
(c) Draw the Gantt chart representation of the project.

32. Consider a software project with 5 tasks T_1–T_5. Duration of the 5 tasks in weeks are 3,2,3,5,2 respectively. T_2 and T_4 can start when T_1 is complete. T_3 can start when T_2 is complete. A T_5 can start when both T_3 and T_4 are complete. Draw the PERT chart representation of the project. When is the latest start date of the task T_3. What is the slack time of the task T4. Which tasks are on the critical path?
33. Explain when should you use PERT charts and when you should use Gantt charts while you are performing the duties of a project manager.
34. How is Gantt chart useful in software project management? What problems might be encountered, if project monitoring and control is carried out using a Gantt chart?
35. Suppose that a certain software product for business application costs Rs. 50,000 to buy off-the-shelf and that its size is 40 KLOC. Assuming that in-house developers cost Rs. 6000 per programmer-month (including overheads), would it be more cost-effective to buy the product or build it? Which elements of the cost are not included in COCOMO estimation model? What additional factors should be considered in making the buy/build decision?
36. What is a baseline in the context of software configuration management? Explain how a baseline can be updated to form a new baseline?
37. Suppose you are the manager of a software project. Explain using only one or two sentences why you should not calculate the number of

developers required for the project as a simple division of the effort estimate (in person-months) by the nominal duration estimate (in months).

38. List the important items that a software project management plan (SPMP) document should discuss.
39. Suppose you are the project manager of a large product development team and you have to make a choice between democratic and chief programmer team organisations, which one would you adopt for your team? Explain the reasoning behind your answer.
40. Compare the relative advantages of the functional and the project approaches of development center organisation. Suppose you are the chief executive officer (CEO) of a software development center. Which organisation structure would you select for your organisation? Why?
41. Name the different ways in which software development teams are organised. For the development of a challenging satellite-based mobile communication product which type of project team organisation would you recommend? Justify your answer.
42. Do you agree with the following statement? "Few, if any, organisation in the real world is purely functional, project, or matrix in nature." Justify your answer.
43. Explain the advantages of a functional organisation over a project organisation. Also explain why software development houses are preferring to use project organisation over functional organisation.
44. In the context of software configuration management, answer the following:
    (a) What do you understand by software configuration?
    (b) What is meant by software configuration management?
    (c) How can you manage software configuration (only mention the names of the principal activities involved)?
    (d) Why is software configuration management crucial to the success of large software product development projects (write only the important reasons)?
    (e) What is change control board (CCB) and what is its role in software configuration management?
45. Why are software projects more susceptible to schedule slippage compared to other types of projects?
46. In what units can you measure the productivity of a software development team? List three important factors that affect the

productivity of a software development team.

47. List three common types of risks that a typical software project might suffer from.
    Explain how you can identify the risks that your project is susceptible to. Suppose you are the project manager of a large software development project, point out the main steps you would follow to manage risks in your software project.
48. Schedule slippage is a very common form of risk that almost every project manager has to encounter. Explain in 3 to 4 sentences how you would manage the risk of schedule slippage as the project manager of a medium-sized project.
49. Explain how you can choose the best risk reduction technique when there are many ways of reducing a risk.
50. What are the important types of risks that a project might suffer from? How would you identify the risks that a project is susceptible to during project the project planning stage?
51. As a project manager, identify the characteristics that you would look for in a software developer while trying to select personnel for your team.
52. What is egoless programming? How can it be realised?
53. Is it true that a software product can always be developed faster by having a larger development team (you can assume that all developers are equally proficient and have exactly similar experience)? Justify your answer.
54. Suppose you have been appointed as the project manager of a large project, identify the activities you would undertake to plan your project. Explain the sequence in which you would undertake these activities by using a task network notation. What are some of the factors which make it hard to accurately estimate the cost of software projects?
55. Suppose you are the project manager of a large development project. The top management informs that you would have to do with a fixed team size (i.e., constant number of developers) through out the duration your project. What will be the impact of this decision on your project?
56. The industry average productivity figure for developers is only 10 LOC/day. What is the reason for such low productivity? Can we attribute this to the poor programming skill of developers?
57. What do you understand by software configuration management? How is it carried out?

What problems would you face if you are developing several versions of the same product according to a client's request, and you are not using any configuration management tools?

58. What is the difference between a revision and a version of a software product? What do you understand by the terms change control and version control? Why are these necessary? Explain how change and version control are achieved using a configuration management tool.
59. Discuss how SCCS or RCS can be used to efficiently manage the configuration of source code. What are some of the shortcomings of SCCS and RCS?
60. Consider a software project with 5 tasks T1-T5. Duration of the 5 tasks (in days) are 15, 10, 12, 25 and 10, respectively. T2 and T4 can start when T1 is complete. T3 can start when T2 is complete. T5 can start when both T3 and T4 are complete. When is the latest start date of the task T3? What is the slack time of the task T4?
61. Why is it necessary for a project manager to decompose the tasks of a project using work breakdown structure (WBS)? To what granularity level are the tasks decomposed? Explain your answer.
62. Suppose you are the project manager of a small team developing a business application.
    Assume that your team has experience in developing several similar products. If you are asked to make a choice between democratic and chief programmer team organisations, which one would you adopt for your team? Explain the reasoning behind your answer.
63. What do you understand by project risk? How can risks be effectively identified by a project manager? How can the risks be managed?
64. Suppose you are appointed as the project manager of a project to develop a commercial word processing software product providing features comparable to MS-WORD software, develop the work breakdown structure (WBS). Explain your answer.
65. What are the different project parameters that determine the cost of a project? What are the important factors which make it hard to accurately estimate the cost of software projects? If you are a project manager bidding for a product development to a customer, would you quote the cost estimated using COCOMO as the price in your bid? Explain your answer.

1 A data processing program is one which processes large volumes of data using a simple algorithm. An example of a data processing application is a payroll software. A payroll software computes the salaries of the employees and prints cheques for them. In a payroll software, the algorithm for pay computation is

fairly simple. The only complexity that arises while developing such a software product is on account of large volumes of data. For example, the pay computation may have to be done for a large number of employees.

2 This section can be skipped for the first-level course on software engineering.

3 Manpower turnover in the software industry parlance is developers leaving an organisation in the middle of a project.

Chapter 4

REQUIREMENTS ANALYSIS AND SPECIFICATION

All plan-driven life cycle models prescribe that before starting to develop a software, the exact requirements of the customer must be understood and documented. In the past, many projects have suffered because the developers started to implement something without determining whether they were building what the customers exactly wanted. Starting development work without properly understanding and documenting the requirements increases the number of iterative changes in the later life cycle phases, and thereby alarmingly pushes up the development costs. This also sets the ground for customer dissatisfaction and bitter customer-developer disputes and protracted legal battles. No wonder that experienced developers consider the requirements analysis and specification to be a very important phase of software development life cycle and undertake it with utmost care.

> Experienced developers take considerable time to understand the exact requirements of the customer and to meticulously document those. They know that without a clear understanding of the problem and proper documentation of the same, it is impossible to develop a satisfactory solution.

For any type of software development project, availability of a good quality requirements document has been acknowledged to be a key factor in the successful completion of the project. A good requirements document not only helps to form a clear understanding of various features required from the software, but also serves as the basis for various activities carried out during later life cycle phases. When software is developed in a contract mode for some other organisation (that is, an outsourced project), the crucial role played by documentation of the precise requirements cannot be overstated. Even when an organisation develops a generic software product, the situation is not very different since some personnel from the organisation's own

marketing department act as the customer. Therefore, for all types of software development projects, proper formulation of requirements and their effective documentation is vital. However, for very small software service projects, the agile methods advocate incremental development of the requirements.

An overview of requirements analysis and specification phase

The requirements analysis and specification phase starts after the feasibility study stage is complete and the project has been found to be financially viable and technically feasible.

The requirements analysis and specification phase ends when the requirements specification document has been developed and reviewed. The requirements specification document is usually called as the *software requirements specification* (SRS) document. The goal of the requirements analysis and specification phase can be stated in a nutshell as follows.

> The goal of the requirements analysis and specification phase is to clearly understand the customer requirements and to systematically organise the requirements into a document called the Software Requirements Specification (SRS) document.

Who carries out requirements analysis and specification?

Requirements analysis and specification activity is usually carried out by a few experienced members of the development team and it normally requires them to spend some time at the customer site. The engineers who gather and analyse customer requirements and then write the requirements specification document are known as *system analysts* in the software industry parlance. System analysts collect data pertaining to the product to be developed and analyse the collected data to conceptualise what exactly needs to be done. After understanding the precise user requirements, the analysts analyse the requirements to weed out inconsistencies, anomalies and incompleteness. They then proceed to write the *software requirements specification* (SRS) document.

> The SRS document is the final outcome of the requirements analysis and specification phase.

How is the SRS document validated?

Once the SRS document is ready, it is first reviewed internally by the

project team to ensure that it accurately captures all the user requirements, and that it is understandable, consistent, unambiguous, and complete. The SRS document is then given to the customer for review. After the customer has reviewed the SRS document and agrees to it, it forms the basis for all future development activities and also serves as a contract document between the customer and the development organisation.

What are the main activities carried out during requirements analysis and specification phase?

Requirements analysis and specification phase mainly involves carrying out the following two important activities:

- Requirements gathering and analysis
- Requirements specification

In the next section, we will discuss the requirements gathering and analysis activity and in the subsequent section we will discuss the requirements specification activity.

4.1 REQUIREMENTS GATHERING AND ANALYSIS

The complete set of requirements are almost never available in the form of a single document from the customer. In fact, it would be unrealistic to expect the customers to produce a comprehensive document containing a precise description of what he wants. Further, the complete requirements are rarely obtainable from any single customer representative. Therefore, the requirements have to be gathered by the analyst from several sources in bits and pieces. These gathered requirements need to be analysed to remove several types of problems that frequently occur in the requirements that have been gathered piecemeal from different sources.

We can conceptually divide the requirements gathering and analysis activity into two separate tasks:

- Requirements gathering
- Requirements analysis

We discuss these two tasks in the following subsections.

4.1.1 Requirements Gathering

Requirements gathering is also popularly known as *requirements elicitation*.

The primary objective of the requirements gathering task is to collect the requirements from the *stakeholders*.

> A stakeholder is a source of the requirements and is usually a person, or a group of persons who either directly or indirectly are concerned with the software.

Requirements gathering may sound like a simple task. However, in practice it is very difficult to gather all the necessary information from a large number of stakeholders and from information scattered across several pieces of documents. Gathering requirements turns out to be especially challenging if there is no working model of the software being developed.

Suppose a customer wants to automate some activity in his organisation that is currently being carried out manually. In this case, a working model of the system (that is, the manual system) exists. Availability of a working model is usually of great help in requirements gathering. For example, if the project involves automating the existing accounting activities of an organisation, then the task of the system analyst becomes a lot easier as he can immediately obtain the input and output forms and the details of the operational procedures. In this context, consider that it is required to develop a software to automate the book-keeping activities involved in the operation of a certain office. In this case, the analyst would have to study the input and output forms and then understand how the outputs are produced from the input data. However, if a project involves developing something new for which no working model exists, then the requirements gathering activity becomes all the more difficult. In the absence of a working system, much more imagination and creativity is required on the part of the system analyst.

Typically even before visiting the customer site, requirements gathering activity is started by studying the existing documents to collect all possible information about the system to be developed. During visit to the customer site, the analysts normally interview the end-users and customer representatives,[1] carry out requirements gathering activities such as questionnaire surveys, task analysis, scenario analysis, and form analysis.

Given that many customers are not computer savvy, they describe their requirements very vaguely. Good analysts share their experience and expertise with the customer and give his suggestions to define certain functionalities more comprehensively, make the functionalities more general and more complete. In the following, we briefly discuss the important ways in which an experienced analyst gathers requirements:

1. Studying existing documentation: The analyst usually studies all the

available documents regarding the system to be developed before visiting the customer site. Customers usually provide statement of purpose (SoP) document to the developers. Typically these documents might discuss issues such as the context in which the software is required, the basic purpose, the stakeholders, features of any similar software developed elsewhere, etc.

2. Interview: Typically, there are many different categories of users of a software. Each category of users typically requires a different set of features from the software. Therefore, it is important for the analyst to first identify the different categories of users and then determine the requirements of each. For example, the different categories of users of a library automation software could be the library members, the librarians, and the accountants. The library members would like to use the software to query availability of books and issue and return books. The librarians might like to use the software to determine books that are overdue, create member accounts, delete member accounts, etc. The accounts personnel might use the software to invoke functionalities concerning financial aspects such as the total fee collected from the members, book procurement expenditures, staff salary expenditures, etc.

To systematise this method of requirements gathering, the Delphi technique can be followed. In this technique, the analyst consolidates the requirements as understood by him in a document and then circulates it for the comments of the various categories of users. Based on their feedback, he refines his document. This procedure is repeated till the different users agree on the set of requirements.

3. Task analysis: The users usually have a black-box view of a software and consider the software as something that provides a set of services (functionalities). A service supported by a software is also called a *task*. We can therefore say that the software performs various tasks of the users. In this context, the analyst tries to identify and understand the different tasks to be performed by the software. For each identified task, the analyst tries to formulate the different steps necessary to realise the required functionality in consultation with the users. For example, for the issue book service, the steps may be—authenticate user, check the number of books issued to the customer and determine if the maximum number of books that this member can borrow has been reached, check whether the book has been reserved, post the book issue details in the member's record, and finally print out a book issue slip that can be presented by the member at the security counter to take the book out of the library premises.

Task analysis helps the analyst to understand the nitty-gritty of various user tasks and to represent each task as a hierarchy of subtasks.

Scenario analysis: A task can have many scenarios of operation. The different scenarios of a task may take place when the task is invoked under different situations. For different types of scenarios of a task, the behaviour of the software can be different. For example, the possible scenarios for the book issue task of a library automation software may be:

- Book is issued successfully to the member and the book issue slip is printed.
- The book is reserved, and hence cannot be issued to the member.
- The maximum number of books that can be issued to the member is already reached, and no more books can be issued to the member.

For various identified tasks, the possible scenarios of execution are identified and the details of each scenario is identified in consultation with the users. For each of the identified scenarios, details regarding system response, the exact conditions under which the scenario occurs, etc. are determined in consultation with the user.

Form analysis: Form analysis is an important and effective requirements gathering activity that is undertaken by the analyst, when the project involves automating an existing manual system. During the operation of a manual system, normally several forms are required to b e filled up by the stakeholders, and in turn they receive several notifications (usually manually filled forms). In form analysis the exiting forms and the formats of the notifications produced are analysed to determine the data input to the system and the data that are output from the system. For the different sets of data input to the system, how these input data would be used by the system to produce the corresponding output data is determined from the users.

Case study 4.1 Requirements gathering for automation of the office work at the CSE department

The academic, inventory, and financial information at the CSE (Computer Science and Engineering) department of a certain institute was being carried out manually by two office clerks, a store keeper, and two attendants. The department has a student strength of 500 and a teacher strength of 30. The head of the department (HoD) wants to automate the office work. Considering the low budget that he has at his disposal, he entrusted the work to a team of student volunteers.

For requirements gathering, a member of the team who was responsible for requirements analysis and specification (analyst) was first briefed by the HoD about the specific activities to be automated. The HoD mentioned that three main aspects of the office work needs to be automated—stores-related activities, student grading activities, and student leave management activities. It was necessary for the analyst to meet the other categories of users. The HoD introduced the analyst (a student) to the office staff. The analyst first discussed with the two clerks regarding their specific responsibilities (tasks) that were required to be automated. For each task, they asked the clerks to brief them about the steps through which these are carried out. The analyst also enquired about the various scenarios that might arise for each task. The analyst collected all types of forms that were being used by the student and the staff of the department to register various types of information with the office (e.g. student course registration, course grading) or requests for some specific service (e.g. issue of items from store). He also collected samples of various types of documents (outputs) the clerks were preparing. Some of these had specific printed forms that the clerks filled up manually, and others were entered using a spreadsheet, and then printed out on a laser printer. For each output form, the analyst consulted the clerks regarding how these different entries are generated from the input data.

The analyst met the store keeper and enquired about the material issue procedures, store ledger entry procedures, and the procedures for raising indents on various vendors. He also collected copies of all the relevant forms that were being used by the store keeper. The analyst also interviewed the student and faculty representatives. Since it was needed to automate the existing activities of an working office, the analyst could without much difficulty obtain the exact formats of the input data, output data, and the precise description of the existing office procedures.

4.1.2 Requirements Analysis

After requirements gathering is complete, the analyst analyses the gathered requirements to form a clear understanding of the exact customer requirements and to weed out any problems in the gathered requirements. It is natural to expect that the data collected from various stakeholders to contain several contradictions, ambiguities, and incompleteness, since each stakeholder typically has only a partial and incomplete view of the software. Therefore, it is necessary to identify all the problems in the requirements and resolve them through further discussions with the customer.

The main purpose of the requirements analysis activity is to analyse the gathered requirements to remove all ambiguities, incompleteness, and inconsistencies from the gathered customer requirements and to obtain a clear understanding of the software to be developed.

For carrying out requirements analysis effectively, the analyst first needs to

develop a clear grasp of the problem. The following basic questions pertaining to the project should be clearly understood by the analyst before carrying out analysis:

- What is the problem?
- Why is it important to solve the problem?
- What exactly are the data input to the system and what exactly are the data output by the system?
- What are the possible procedures that need to be followed to solve the problem?
- What are the likely complexities that might arise while solving the problem?
- If there are external software or hardware with which the developed software has to interface, then what should be the data interchange formats with the external systems?

After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various problems that he detects in the gathered requirements.

During requirements analysis,the analyst needs to identify and resolve three main types of problems in the requirements:
- Anomaly
- Inconsistency
- Incompleteness

Let us examine these different types of requirements problems in detail.

Anomaly: It is an anomaly is an ambiguity in a requirement. When a requirement is anomalous, several interpretations of that requirement are possible. Any anomaly in any of the requirements can lead to the development of an incorrect system, since an anomalous requirement can be interpreted in the several ways during development. The following are two examples of anomalous requirements:

Example 4.1 While gathering the requirements for a process control application, the following requirement was expressed by a certain stakeholder: When the temperature becomes high, the heater should be switched off. Please note that words such as "high", "low", "goo d", "bad" etc. are indications of ambiguous requirements as these lack quantification and can be subjectively interpreted. If the threshold above which the temperature

can be considered to be high is not specified, then it can be interpreted differently by different developers.

Example 4.2 In the case study 4.1, suppose one office clerk described the following requirement: during the final grade computation, if any student scores a sufficiently low grade in a semester, then his parents would need to be informed. This is clearly an ambiguous requirement as it lacks any well defined criterion as to what can be considered as a "sufficiently low grade".

Inconsistency: Two requirements are said to be inconsistent, if one of the requirements contradicts the other. The following are two examples of inconsistent requirements:

Example 4.3 Consider the following two requirements that were collected from two different stakeholders in a process control application development project.

- The furnace should be switched-off when the temperature of the furnace rises above 500□ C.
- When the temperature of the furnace rises above 500□ C, the water shower should be switched-on and the furnace should remain on.

The requirements expressed by the two stakeholders are clearly inconsistent.

Example 4.4 In the case study 4.1 suppose one of the clerks gave the following requirement— a student securing fail grades in three or more subjects must repeat the courses over an entire semester, and he cannot credit any other courses while repeating the courses. Suppose another clerk expressed the following requirement—there is no provision for any student to repeat a semester; the student should clear the subject by taking it as an extra subject in any later semester. There is a clear inconsistency between the requirements given by the two stakeholders.

Incompleteness: An incomplete set of requirements is one in which some requirements have been overlooked. The lack of these features would be felt by the customer much later, possibly while using the software. Often, incompleteness is caused by the inability of the customer to visualise the system that is to be developed and to anticipate all the features that would be required. An experienced analyst can detect most of these missing features and suggest them to the customer for his consideration and approval for incorporation in the requirements. The following are two examples of

incomplete requirements:

Example 4.5 Suppose for the case study 4.1, one of the clerks expressed the following—If a student secures a *grade point average* (GPA) of less than 6, then the parents of the student must be intimated about the regrettable performance through a (postal) letter as well as through e-mail. However, on an examination of all requirements, it was found that there is no provision by which either the postal or e-mail address of the parents of the students can be entered into the system. The feature that would allow entering the e-mail ids and postal addresses of the parents of the students was missing, thereby making the requirements incomplete.

Example 4.6 In a chemical plant automation software, suppose one of the requirements is that if the internal temperature of the reactor exceeds 200 □C then an alarm bell must be sounded. However, on an examination of all requirements, it was found that there is no provision for resetting the alarm bell after the temperature has been brought down in any of the requirements. This is clearly an incomplete requirement.

Can an analyst detect all the problems existing in the gathered requirements?

Many of the inconsistencies, anomalies, and incompleteness are detected effortlessly, while some others require a focused study of the specific requirements. A few problems in the requirements can, however, be very subtle and escape even the most experienced eyes. Many of these subtle anomalies and inconsistencies can be detected, if the requirements are specified and analysed using a formal method. Once a system has been formally specified, it can be systematically (and even automatically) analysed to remove all problems from the specification. We will discuss the basic concepts of formal system specification in Section 4.3. Though the use of formal techniques is not widespread, the current practice is to formally specify only the safety-critical[2] parts of a system.

4.2 SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

After the analyst has gathered all the required information regarding the software to be developed, and has removed all incompleteness, inconsistencies, and anomalies from the specification, he starts to systematically organise the requirements in the form of an SRS

document. The SRS document usually contains all the user requirements in a structured though an informal form.

Among all the documents produced during a software development life cycle, SRS document is probably the most important document and is the toughest to write. One reason for this difficulty is that the SRS document is expected to cater to the needs of a wide variety of audience. In the following subsection, we discuss the different categories of users of an SRS document and their needs from it.

4.2.1 Users of SRS Document

Usually a large number of different people need the SRS document for very different purposes. Some of the important categories of users of the SRS document and their needs for use are as follows:

Users, customers, and marketing personnel: These stakeholders need to refer to the SRS document to ensure that the system as described in the document will meet their needs. Remember that the customer may not be the user of the software, but may be some one employed or designated by the user. For generic products, the marketing personnel need to understand the requirements that they can explain to the customers.

Software developers: The software developers refer to the SRS document to make sure that they are developing exactly what is required by the customer.

Test engineers: The test engineers use the SRS document to understand the functionalities, and based on this write the test cases to validate its working. They need that the required functionality should be clearly described, and the input and output data should have been identified precisely.

User documentation writers: The user documentation writers need to read the SRS document to ensure that they understand the features of the product well enough to be able to write the users' manuals.

Pro ject managers: The project managers refer to the SRS document to ensure that they can estimate the cost of the project easily by referring to the SRS document and that it contains all the information required to plan the project.

Maintenance engineers: The SRS document helps the maintenance engineers to under- stand the functionalities supported by the system. A clear knowledge of the functionalities can help them to understand the design and

code. Also, a proper understanding of the functionalities supported enables them to determine the specific modifications to the system's functionalities would be needed for a specific purpose.

Many software engineers in a project consider the SRS document to be a reference document. However, it is often more appropriate to think of the SRS document as the documentation of a contract between the development team and the customer. In fact, the SRS document can be used to resolve any disagreements between the developers and the customers that may arise in the future. The SRS document can even be used as a legal document to settle disputes between the customers and the developers in a court of law. Once the customer agrees to the SRS document, the development team proceeds to develop the software and ensure that it conforms to all the requirements mentioned in the SRS document.

4.2.2 Why Spend Time and Resource to Develop an SRS Document?

A well-formulated SRS document finds a variety of usage other than the primary intended usage as a basis for starting the software development work. In the following subsection, we identify the important uses of a well-formulated SRS document:

Forms an agreement between the customers and the developers: A good SRS document sets the stage for the customers to form their expectation about the software and the developers about what is expected from the software.

Reduces future reworks: The process of preparation of the SRS document forces the stakeholders to rigorously think about all of the requirements before design and development get underway. This reduces later redesign, recoding, and retesting. Careful review of the SRS document can reveal omissions, misunderstandings, and inconsistencies early in the development cycle.

Provides a basis for estimating costs and schedules: Project managers usually estimate the size of the software from an analysis of the SRS document. Based on this estimate they make other estimations such as the effort required to develop the software and the total cost of development. The SRS document also serves as a basis for price negotiations with the customer. The project manager also uses the SRS document for work scheduling.

Provides a baseline for validation and verification: The SRS document

provides a baseline against which compliance of the developed software can be checked. It is also used by the test engineers to create the *test plan*.

Facilitates future extensions: The SRS document usually serves as a basis for planning future enhancements.

Before we discuss about how to write an SRS document, we first discuss the characteristics of a good SRS document and the pitfalls that one must consciously avoid while writing an SRS document.

4.2.3 Characteristics of a Good SRS Document

The skill of writing a good SRS document usually comes from the experience gained from writing SRS documents for many projects. However, the analyst should be aware of the desirable qualities that every good SRS document should possess. IEEE Recommended Practice for Software Requirements Specifications[IEEE830] describes the content and qualities of a good software requirements specification (SRS). Some of the identified desirable qualities of an SRS document are the following:

- **Concise:** The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase the possibilities of errors in the document.
- **Implementation-independent:** The SRS should be free of design and implementation decisions unless those decisions reflect actual requirements. It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the externally visible behaviour of the system and not discuss the implementation issues. This view with which a requirements specification is written, has been shown in Figure 4.1. Observe that in Figure 4.1, the SRS document describes the output produced for the different types of input and a description of the processing required to produce the output from the input (shown in ellipses) and the internal working of the software is not discussed at all.

The SRS document should describe the system to be developed as a black box, and should specify only the externally visible behaviour of the system. For this reason, the SRS document is also called the *black-box* specification of the software being

developed.

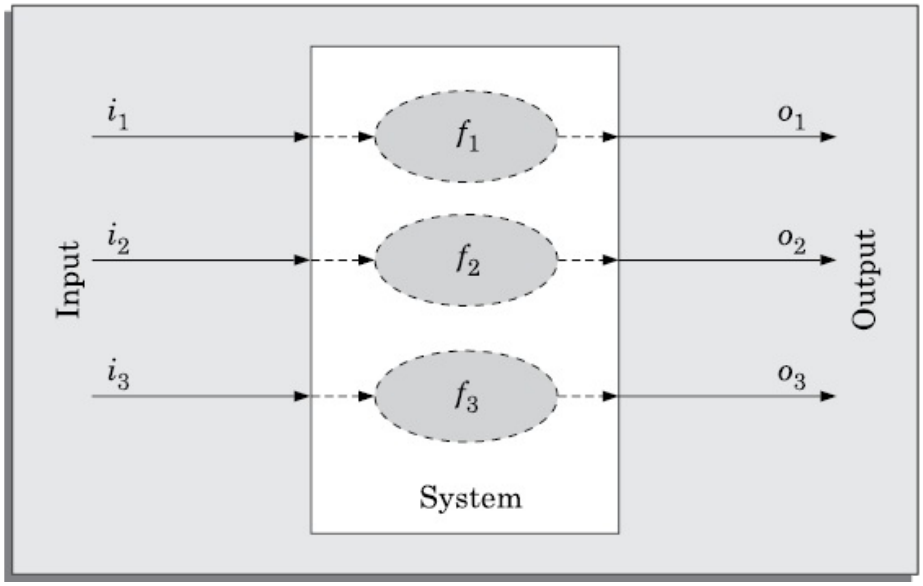

Figure 4.1: The black-box view of a system as performing a set of functions.

Traceable: It should be possible to trace a specific requirement to the design elements that implement it and *vice versa*. Similarly, it should be possible to trace a requirement to the code segments that implement it and the test cases that test this requirement and *vice versa*. Traceability is also important to verify the results of a phase with respect to the previous phase and to analyse the impact of changing a requirement on the design elements and the code.

Modifiable: Customers frequently change the requirements during the software development development due to a variety of reasons. Therefore, in practice the SRS document undergoes several revisions during software development. Also, an SRS document is often modified after the project completes to accommodate future enhancements and evolution. To cope up with the requirements changes, the SRS document should be easily modifiable. For this, an SRS document should be well-structured. A well-structured document is easy to understand and modify. Having the description of a requirement scattered across many places in the SRS document may not be wrong—but it tends to make the requirement difficult to understand and also any modification to the requirement would become difficult as it would require changes to be made at large number of places in the document.

Identification of response to undesired events: The SRS document should discuss the system responses to various undesired events and

exceptional conditions that may arise.

Verifiable: All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to design test cases based on the description of the functionality as to whether or not requirements have been met in an implementation. A requirement such as "the system should be user friendly" is not verifiable. On the other hand, the requirement—"When the name of a book is entered, the software should display whether the book is available for issue or it has been loaned out" is verifiable. Any feature of the required system that is not verifiable should be listed separately in the goals of the implementation section of the SRS document.

4.2.4 Attributes of Bad SRS Documents

SRS documents written by novices frequently suffer from a variety of problems. As discussed earlier, the most damaging problems are incompleteness, ambiguity, and contradictions. There are many other types problems that a specification document might suffer from. By knowing these problems, one can try to avoid them while writing an SRS document. Some of the important categories of problems that many SRS documents suffer from are as follows:

Over-specification: It occurs when the analyst tries to address the "how to" aspects in the SRS document. For example, in the library automation problem, one should not specify whether the library membership records need to be stored indexed on the member's first name or on the library member's identification (ID) number. Over-specification restricts the freedom of the designers in arriving at a good design solution.

Forward references: One should not refer to aspects that are discussed much later in the SRS document. Forward referencing seriously reduces readability of the specification.

Wishful thinking: This type of problems concern description of aspects which would be difficult to implement.

Noise: The term noise refers to presence of material not directly relevant to the software development process. For example, in the register customer function, suppose the analyst writes that customer registration department is manned by clerks who report for work between 8am and 5pm, 7 days a week. This information can be called *noise* as it would hardly be of any use to the software developers and would unnecessarily clutter the SRS document,

diverting the attention from the crucial points.

Several other "sins" of SRS documents can be listed and used to guard against writing a bad SRS document and is also used as a checklist to review an SRS document.

4.2.5 Important Categories of Customer Requirements

A good SRS document, should properly categorize and organise the requirements into different sections [IEEE830]. As per the IEEE 830 guidelines, the important categories of user requirements are the following.

An SRS document should clearly document the following aspects of a software:
- Functional requirements
- Non-functional requirements
  - Design and implementation constraints
  - External interfaces required
  - Other non-functional requirements
- Goals of implementation.

In the following subsections, we briefly describe the different categories of requirements.

Functional requirements

The functional requirements capture the functionalities required by the users from the system. We have already pointed out in Chapter 2 that it is useful to consider a software as offering a set of functions $\{f_i\}$ to the user. These functions can be considered similar to a mathematical function $f : I \rightarrow O$, meaning that a function transforms an element (i_i) in the input domain (I) to a value (o_i) in the output (O). This functional view of a system is shown schematically in Figure 4.1. Each function f_i of the system can be considered as reading certain data i_i, and then transforming a set of input data (i_i) to the corresponding set of output data (o_i). The functional requirements of the system, should clearly describe each functionality that the system would support along with the corresponding input and output data set. Considering that the functional requirements are a crucial part of the SRS document, we discuss functional requirements in more detail in Section 4.2.6. Section 4.2.7 discusses how the functional requirements can be identified from a problem description.

Finally, Section 4.2.8 discusses how the functional requirements can be

documented effectively.

Non-functional requirements

The non-functional requirements are non-negotiable obligations that must be supported by the software. The non-functional requirements capture those requirements of the customer that cannot be expressed as functions (i.e., accepting input data and producing output data). Non-functional requirements usually address aspects concerning external interfaces, user interfaces, maintainability, portability, usability, maximum number of concurrent users, timing, and throughput (transactions per second, etc.). The non-functional requirements can be critical in the sense that any failure by the developed software to achieve some minimum defined level in these requirements can be considered as a failure and make the software unacceptable by the customer.

> The IEEE 830 standard recommends that out of the various non-functional requirements, the external interfaces, and the design and implementation constraints should be documented in two different sections. The remaining non-functional requirements should be documented later in a section and these should include the performance and security requirements.

In the following subsections, we discuss the different categories of non-functional requirements that are described under three different sections:

Design and implementation constraints: Design and implementation constraints are an important category of non-functional requirements describe any items or issues that will limit the options available to the developers. Some of the example constraints can be—corporate or regulatory policies that needs to be honoured; hardware limitations; interfaces with other applications; specific technologies, tools, and databases to be used; specific communications protocols to be used; security considerations; design conventions or programming standards to be followed, etc. Consider an example of a constraint that can be included in this section—Oracle DBMS needs to be used as this would facilitate easy interfacing with other applications that are already operational in the organisation.

External interfaces required: Examples of external interfaces are—hardware, software and communication interfaces, user interfaces, report formats, etc. To specify the user interfaces, each interface between the software and the users must be described. The description may include sample screen images, any GUI standards or style guides that are to be

followed, screen layout constraints, standard buttons and functions (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, and so on. One example of a user interface requirement of a software can be that it should be usable by factory shop floor workers who may not even have a high school degree. The details of the user interface design such as screen designs, menu structure, navigation diagram, etc. should be documented in a separate user interface specification document.

Other non-functional requirements: This section contains a description of non- functional requirements that are neither design constraints and nor are external interface requirements. An important example is a performance requirement such as the number of transactions completed per unit time. Besides performance requirements, the other non-functional requirements to be described in this section may include reliability issues, accuracy of results, and security issues.

Goals of implementation

The 'goals of implementation' part of the SRS document offers some general suggestions regarding the software to be developed. These are not binding on the developers, and they may take these suggestions into account if possible. For example, the developers may use these suggestions while choosing among different design solutions.

A goal, in contrast to the functional and non-functional requirements, is not checked by the customer for conformance at the time of acceptance testing.

The goals of implementation section might document issues such as easier revisions to the system functionalities that may be required in the future, easier support for new devices to be supported in the future, reusability issues, etc. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately. It is useful to remember that anything that would be tested by the user and the acceptance of the system would depend on the outcome of this task, is usually considered as a requirement to be fulfilled by the system and not a goal and *vice versa*.

How to classify the different types of requirements?

We should be clear regarding the aspects of the system requirement that are to be documented as the functional requirement, the ones to be documented as non-functional requirement, and the ones to be

documented as the goals of implementation. Aspects which can be expressed as transformation of some input data to some output data (i.e., the functions of the system) should be documented as the functional requirement. Any other requirements whose compliance by the developed system can be verified by inspecting the system are documented as non- functional requirements. Aspects whose compliance by the developed system need not be verified but are merely included as suggestions to the developers are documented as goals of the implementation.

The difference between non-functional requirements and guidelines is the following. Non-functional requirements would be tested for compliance, before the developed product is accepted by the customer whereas guideline, on the other hand, are customer request that are desirable to be done, but would not be tested during product acceptance.

Functional requirements form the basis for most design and test methodologies. Therefore, unless the functional requirements are properly identified and documented, the design and testing activities cannot be carried out satisfactorily. We discuss how to document the functional requirements in the next section.

4.2.6 Functional Requirements

In order to document the functional requirements of a system, it is necessary to first learn to identify the high-level functions of the systems by reading the informal documentation of the gathered requirements. The high-level functions would be split into smaller subrequirements. Each high-level function is an instance of use of the system (use case) by the user in some way.

A high-level function is one using which the user can get some useful piece of work done.

However, the above is not a very accurate definition of a high-level function. For example, how useful must a piece of work be performed by the system for it to be called 'a useful piece of work' ? Can the printing of the statements of the ATM transaction during withdrawal of money from an ATM be called a useful piece of work? Printing of ATM transaction should not be considered a high-level requirement, because the user does not specifically request for this activity. The receipt gets printed automatically as part of the withdraw money function. Usually, the user invokes (requests) the services of each high-level requirement. It may therefore be possible to treat print

receipt as part of the withdraw money function rather than treating it as a high-level function. It is therefore required that for some of the high-level functions, we might have to debate whether we wish to consider it as a high-level function or not. However, it would become possible to identify most of the high-level functions without much difficulty after practising the solution to a few exercise problems.

Each high-level requirement typically involves accepting some data from the user through a user interface, transforming it to the required response, and then displaying the system response in proper format. For example, in a library automation software, a high-level functional requirement might be search-book. This function involves accepting a book name or a set of key words from the user, running a matching algorithm on the book list, and finally outputting the matched books. The generated system response can be in several forms, e.g., display on the terminal, a print out, some data transferred to the other systems, etc. However, in degenerate cases, a high-level requirement may not involve any data input to the system or production of displayable results. For example, it may involve switch on a light, or starting a motor in an embedded application.

Are high-level functions of a system similar to mathematical functions?

We all know that a mathematical function transforms input data to output data. A high-level function transforms certain input data to output data. However, except for very simple high- level functions, a function rarely reads all its required data in one go and rarely outputs all the results in one shot. In fact, a high-level function usually involves a series of interactions between the system and one or more users. An example of the interactions that may occur in a single high-level requirement has been shown in Figure 4.2. In Figure 4.2, the user inputs have been represented by rectangles and the response produced by the system by circles. Observe that the rectangles and circles alternate in the execution of a single high-level function of the system, indicating a series of requests from the user and the corresponding responses from the system. Typically, there is some initial data input by the user. After accepting this, the system may display some response (called *system action*). Based on this, the user may input further data, and so on.

For any given high-level function, there can be different interaction

sequences or scenarios due to users selecting different options or entering different data items.

In Figure 4.2, the different scenarios occur depending on the amount entered for withdrawal. The different scenarios are essentially different behaviour exhibited by the system for the same high-level function. Typically, each user input and the corresponding system action may be considered as a sub-requirement of a high-level requirement. Thus, each high-level requirement can consist of several sub-requirements.

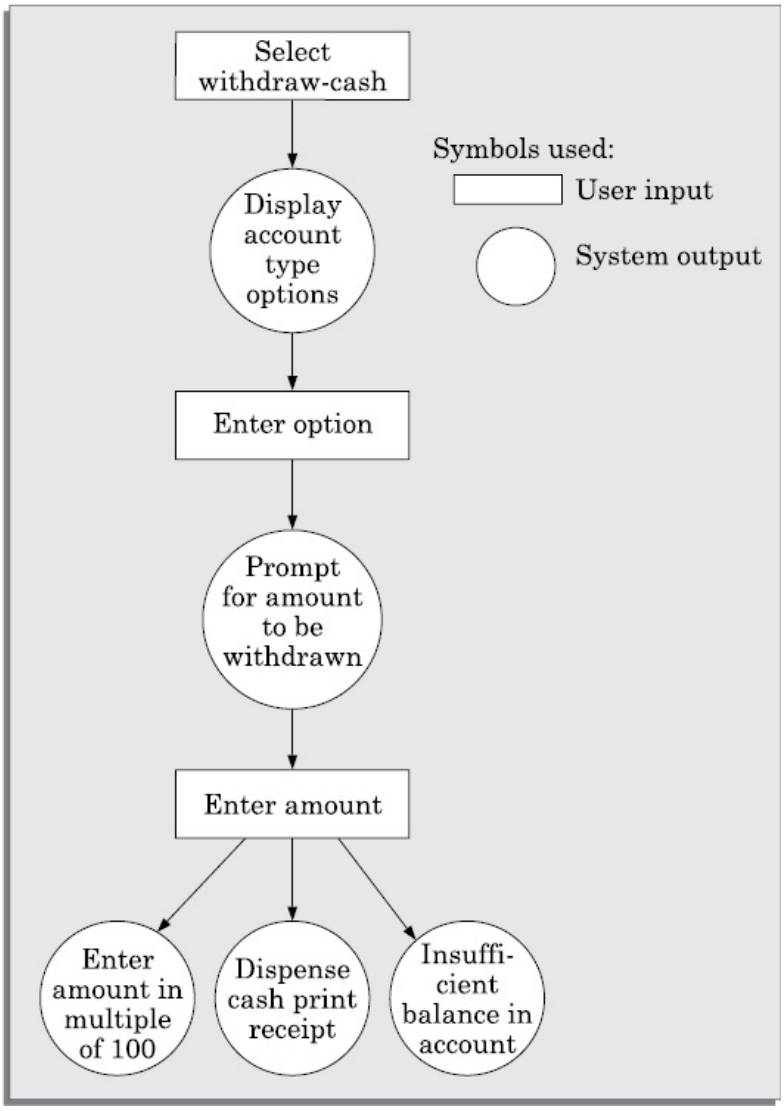

Figure 4.2: User and system interactions in high-level functional requirement.

Is it possible to determine all input and output data precisely?

In a requirements specification document, it is desirable to define the precise data input to the system and the precise data output by the system. Sometimes, the exact data items may be very difficult to

identify. This is especially the case, when no working model of the system to be developed exists. In such cases, the data in a high-level requirement should be described using high-level terms and it may be very difficult to identify the exact components of this data accurately. Another aspect that must be kept in mind is that the data might be input to the system in stages at different points in execution. For example, consider the withdraw-cash function of an *automated teller machine* (ATM) of Figure 4.2. Since during the course of execution of the withdraw-cash function, the user would have to input the type of account, the amount to be withdrawn, it is very difficult to form a single high-level name that would accurately describe both the input data. However, the input data for the subfunctions can be more accurately described.

4.2.7 How to Identify the Functional Requirements?

The high-level functional requirements often need to be identified either from an informal problem description document or from a conceptual understanding of the problem.

Each high-level requirement characterises a way of system usage (service invocation) by some user to perform some meaningful piece of work.

Remember that there can be many types of users of a system and their requirements from the system may be very different. So, it is often useful to first identify the different types of users who might use the system and then try to identify the different services expected from the software by different types of users.

The decision regarding which functionality of the system can be taken to be a high-level functional requirement and the one that can be considered as part of another function (that is, a subfunction) leaves scope for some subjectivity. For example, consider the issue-book function in a Library Automation System. Suppose, when a user invokes the issue-book function, the system would require the user to enter the details of each book to be issued. Should the entry of the book details be considered as a high-level function, or as only a part of the issue-book function? Many times, the choice is obvious. But, sometimes it requires making non-trivial decisions.

4.2.8 How to Document the Functional Requirements?

Once all the high-level functional requirements have been identified and

the requirements problems have been eliminated, these are documented. A function can be documented by identifying the state at which the data is to be input to the system, its input data domain, the output data domain, and the type of processing to be carried on the input data to obtain the output data. We now illustrate the specification of the functional requirements through two examples. Let us first try to document the withdraw-cash function of an *automated teller machine* (ATM) system in the following. The withdraw-cash is a high-level requirement. It has several sub-requirements corresponding to the different user interactions. These user interaction sequences may vary from one invocation from another depending on some conditions. These different interaction sequences capture the different *scenarios*. To accurately describe a functional requirement, we must document all the different scenarios that may occur.

Example 4.7 (Withdraw cash from ATM): An initial informal description of a required functionality is usually given by the customer as a *statement of purpose* (SoP). An SoP serves as a starting point for the analyst and he proceeds with the requirements gathering activity after a basic understanding of the SoP. However, the functionalities of withdraw cash from ATM is intuitively obvious to any one who has used a bank ATM. So, we are not including an informal description of withdraw cash functionality here and in the following, we documents this functional requirement.

R.1: Withdraw cash

Description: The withdraw cash function first determines the type of account that the user has and the account number from which the user wishes to withdraw cash. It checks the balance to determine whether the requested amount is available in the account. If enough balance is available, it outputs the required cash, otherwise it generates an error message.

R.1.1: Select withdraw amount option

Input: "Withdraw amount" option selected *Output:* User prompted to enter the account type

R.1.2: Select account type

Input: User selects option from any one of the followings—savings/checking/deposit.

Output: Prompt to enter amount

R.1.3: Get required amount

Input: Amount to be withdrawn in integer values greater than 100 and less than 10,000 in multiples of 100.

Output: The requested cash and printed transaction statement.

Processing: The amount is debited from the user's account if sufficient balance is available, otherwise an error message displayed.

Example 4.8 (Search book availability in library): An initial informal description of a required functionality is usually given by the customer as a *statement of purpose* (SoP) based on which an later requirements gathering, the analyst understand the functionality. However, the functionalities of *search book availability* is intuitively obvious to any one who has used a library. So, we are not including an informal description of *searc h book availability* functionality here and in the following, we documents this functional requirement.

R.1: Search book

Description Once the user selects the search option, he would be asked to enter the keywords. The system would search the book in the book list based on the key words entered. After making the search, the system should output the details of all books whose title or author name match any of the key words entered. The book details to be displayed include: title, author name, publisher name, year of publication, ISBN number, catalog number, and the location in the library.

R.1.1: Select search option

Input: "Search" option

Output: User prompted to enter the key words

R.1.2: Search and display

Input: Key words

Output: Details of all books whose title or author name matches any of the key words entered by the user. The book details displayed would include—title of the book, author name, ISBN number, catalog number, year of publication, number of copies available, and the location in the library.

Processing: Search the book list based on the key words:

R.2: Renew book

Description: When the "renew" option is selected, the user is asked to enter his membership number and password. After password validation, the list of

the books borrowed by him are displayed. The user can renew any of his borrowed books by indicating them. A requested book cannot be renewed if it is reserved by another user. In this case, an error message would be displayed.

R.2.1: Select renew option

State: The user has logged in and the main menu has been displayed.

Input: "Renew" option selection.

Output: Prompt message to the user to enter his membership number and password.

R.2.2: Login

State: The renew option has been selected.

Input: Membership number and password.

Output: List of the books borrowed by the user is displayed, and user is prompted to select the books to be renewed, if the password is valid. If the password is invalid, the user is asked to re-enter the password.

Processing: Password validation, search the books issued to the user from the borrower's list and display.

Next function: R.2.3 if password is valid and R.2.2 if password is invalid.

R.2.3: Renew selected books

Input: User choice for books to be renewed out of the books borrowed by him.

Output: Confirmation of the books successfully renewed and apology message for the books that could not be renewed.

Processing: Check if any one has reserved any of the requested books. Renew the books selected by the user in the borrower's list, if no one has reserved those books.

In order to properly identify the high-level requirements, a lot of common sense and the ability to visualise various scenarios that might arise in the operation of a function are required. Please note that when any of the aspects of a requirement, such as the state, processing description, next function to be executed, etc. are obvious, we have omitted it. We have to make a trade-off between cluttering the document with trivial details versus missing out some important descriptions.

Specification of large software: If there are large number of functional requirements (much larger than seen), should they just be written in a long numbered list of requirements? A better way to organise the functional

requirements in this case would be to split the requirements into sections of related requirements. For example, the functional requirements of a academic institute automation software can be split into sections such as accounts, academics, inventory, publications, etc. When there are too many functional requirements, these should be properly arranged into sections. For example the following can be sections in the trade house automation software:

- Customer management
- Account management
- Purchase management
- Vendor management
- Inventory management

Level of details in specification: Even for experienced analysts, a common dilemma is in specifying too little or specifying too much. In practice, we would have to specify only the important input/output interactions in a functionality along with the processing required to generate the output from the input. However, if the interaction sequence is specified in too much detail, then it becomes an unnecessary constraint on the developers and restricts their choice in solution. On the other hand, if the interaction sequence is not sufficiently detailed, it may lead to ambiguities and result in improper implementation.

4.2.9 Traceability

Traceability means that it would be possible to identify (trace) the specific design component which implements a given requirement, the code part that corresponds to a given design component, and test cases that test a given requirement. Thus, any given code component can be traced to the corresponding design component, and a design component can be traced to a specific requirement that it implements and *vice versa*. Traceability analysis is an important concept and is frequently used during software development. For example, by doing a traceability analysis, we can tell whether all the requirements have been satisfactorily addressed in all phases. It can also be used to assess the impact of a requirements change. That is, traceability makes it easy to identify which parts of the design and code would be affected, when certain requirement change occurs. It can also be used to study the impact of a bug that is known to exist in a code part on various

requirements, etc.

To achieve traceability, it is necessary that each functional requirement should be numbered uniquely and consistently. Proper numbering of the requirements makes it possible for different documents to uniquely refer to specific requirements. An example scheme of numbering the functional requirements is shown in Examples 4.7 and 4.8, where the functional requirements have been numbered R.1, R.2, etc. and the subrequirements for the requirement R.1 have been numbered R.1.1, R.1.2, etc.

4.2.10 Organisation of the SRS Document

In this section, we discuss the organisation of an SRS document as prescribed by the IEEE 830 standard[IEEE 830]. Please note that IEEE 830 standard has been intended to serve only as a guideline for organizing a requirements specification document into sections and allows the flexibility of tailoring it, as may be required for specific projects. Depending on the type of project being handled, some sections can be omitted, introduced, or interchanged as may be considered prudent by the analyst. However, organisation of the SRS document to a large extent depends on the preferences of the system analyst himself, and he is often guided in this by the policies and standards being followed by the development company. Also, the organisation of the document and the issues discussed in it to a large extent depend on the type of the product being developed. However, irrespective of the company's principles and product type, the three basic issues that any SRS document should discuss are—functional requirements, non-functional requirements, and guidelines for system implementation.

The introduction section should describe the context in which the system is being developed, and provide an overall description of the system, and the environmental characteristics. The introduction section may include the hardware that the system will run on, the devices that the system will interact with and the user skill-levels. Description of the user skill-level is important, since the command language design and the presentation styles of the various documents depend to a large extent on the types of the users it is targeted for. For example, if the skill-levels of the users is "novice", it would mean that the user interface has to be very simple and rugged, whereas if the user-level is "advanced", several short cut techniques and advanced features may be provided in the user interface.

It is desirable to describe the formats for the input commands, input data, output reports, and if necessary the modes of interaction. We have already

discussed how the contents of the Sections on the functional requirements, the non-functional requirements, and the goals of implementation should be written. In the following subsections, we outline the important sections that an SRS document should contain as suggested by the IEEE 830 standard, for each section of the document, we also briefly discuss the aspects that should be discussed in it.

Introduction

Purpose: This section should describe where the software would be deployed and and how the software would be used.

Project scope: This section should briefly describe the overall context within which the software is being developed. For example, the parts of a problem that are being automated and the parts that would need to be automated during future evolution of the software.

Environmental characteristics: This section should briefly outline the environment (hardware and other software) with which the software will interact.

Overall description of organisation of SRS document

Product perspective: This section needs to briefly state as to whether the software is intended to be a replacement for a certain existing systems, or it is a new software. If the software being developed would be used as a component of a larger system, a simple schematic diagram can be given to show the major components of the overall system, subsystem interconnections, and external interfaces can be helpful.

Product features: This section should summarize the major ways in which the software would be used. Details should be provided in Section 3 of the document. So, only a brief summary should be presented here.

User classes: Various user classes that are expected to use this software are identified and described here. The different classes of users are identified by the types of functionalities that they are expected to invoke, or their levels of expertise in using computers.

Operating environment: This section should discuss in some detail the hardware platform on which the software would run, the operating system, and other application software with which the developed software would interact.

Design and implementation constraints: In this section, the different

constraints on the design and implementation are discussed. These might include—corporate or regulatory policies; hardware limitations (timing requirements, memory requirements); interfaces to other applications; specific technologies, tools, and databases to be used; specific programming language to be used; specific communication protocols to be used; security considerations; design conventions or programming standards.

User documentation: This section should list out the types of user documentation, such as user manuals, on-line help, and trouble-shooting manuals that will be delivered to the customer along with the software.

Functional requirements for organisation of SRS document

This section can classify the functionalities either based on the specific functionalities invoked by different users, or the functionalities that are available in different modes, etc., depending what may be appropriate.

1. User class 1
   (a) Functional requirement 1.1
   (b) Functional requirement 1.2
2. User class 2
   (a) Functional req uirement 2.1
   (b) Functional requirement 2.2

External interface requirements

User interfaces: This section should describe a high-level description of various interfaces and various principles to be followed. The user interface description may include sample screen images, any GUI standards or style guides that are to be followed, screen layout constraints, standard push buttons (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, etc. The details of the user interface design should be documented in a separate user interface specification document.

Hardware interfaces: This section should describe the interface between the software and the hardware components of the system. This section may include the description of the supported device types, the nature of the data and control interactions between the software and the hardware, and the communication protocols to be used.

Software interfaces: This section should describe the connections between this software and other specific software components, including databases,

operating systems, tools, libraries, and integrated commercial components, etc. Identify the data items that would be input to the software and the data that would be output should be identified and the purpose of each should be described.

Communications interfaces: This section should describe the requirements associated with any type of communications required by the software, such as e-mail, web access, network server communications protocols, etc. This section should define any pertinent message formatting to be used. It should also identify any communication standards that will be used, such as TCP sockets, FTP, HTTP, or SHTTP. Specify any com munication security or encryption issues that may be relevant, and also the data transfer rates, and synchronisation mechanisms.

Other non-functional requirements for organisation of SRS document

This section should describe the non-functional requirements other than the design and implementation constraints and the external interface requirements that have been described in Sections 2 and 4 respectively.

Performance requirements: Aspects such as number of transaction to be completed per second should be specified here. Some performance requirements may be specific to individual functional requirements or features. These should also be specified here.

Safety requirements: Those requirements that are concerned with possible loss or damage that could result from the use of the software are specified here. For example, recovery after power failure, handling software and hardware failures, etc. may be documented here.

Security requirements: This section should specify any requirements regarding security or privacy requirements on data used or created by the software. Any user identity authentication requirements should be described here. It should also refer to any external policies or regulations concerning the security issues. Define any security or privacy certifications that must be satisfied.

For software that have distinct modes of operation, in the functional requirements section, the different modes of operation can be listed and in each mode the specific functionalities that are available for invocation can be organised as follows.

Functional requirements

1. Operation mode 1
   (a) Functional requirement 1.1
   (b) Functional requirement 1.2
2. Operation mode 2
   (a) Functional requirement 2.1
   (b) Functional requirement 2.2

Specification of the behaviour may not be necessary for all systems. It is usually necessary for those systems in which the system behaviour depends on the state in which the system is, and the system transits among a set of states depending on some prespecified conditions and events. The behaviour of a system can be specified using either the *finite state machine* (FSM) formalism and any other alternate formalisms. The FSMs can used to specify the possible states (modes) of the system and the transition among these states due to occurrence of events.

Example 4.9 (Personal library software): It is proposed to develop a software that would be used by individuals to manage their personal collection of books. The following is an informal description of the requirements of this software as worked out by the marketing department. Develop the functional and non-functional requirements for the software.

A person can have up to a few hundreds of books. The details of all the books such as name of the book, year of publication, date of purchase, price, and publisher would be entered by the owner. A book should be assigned a unique serial number by the computer. This number would be written by the owner using a pen on the inside page of the book. Only a registered friend can be lent a book. While registering a friend, the following data would have to be supplied—name of the friend, his address, land line number, and mobile number. Whenever a book issue request is given, the name of the friend to whom the book is to be issued and the unique id of the book is entered. At this, the various books outstanding against the borrower along with the date borrowed are displayed for information of the owner. If the owner wishes to go ahead with the issue of the book, then the date of issue, the title of the book, and the unique identification number of the book are stored. When a friend returns a book, the date of return is stored and the book is removed from his borrowing list. Upon query, the software should display the name, address, and telephone numbers of each friend against whom books are outstanding along with the titles of the outstanding books and the date on

which those were issued. The software should allow the owner to update the details of a friend such as his address, phone, telephone number, etc. It should be possible for the owner to delete all the data pertaining to a friend who is no more active in using the library. The records should be stored using a free (public domain) data base management system. The software should run on both Windows and Unix machines.

Whenever the owner of the library software borrows a book from his friends, would enter the details regarding the title of the book, and the date borrowed and the friend from whom he borrowed it. Similarly, the return details of books would be entered. The software should be able to display all the books borrowed from various friends upon request by the owner.

It should be possible for any one to query about the availability of a particular book through a web browser from any location. The owner should be able to query the total number of books in the personal library, and the total amount he has invested in his library. It should also be possible for him to view the number of books borrowed and returned by any (or all) friend(s) over any specified time.

Functional requirements

The software needs to support three categories of functionalities as described below:

1. Manage own books

1.1 Register book

Description: To register a book in the personal library, the details of a book, such as name, year of publication, date of purchase, price and publisher are entered. This is stored in the database and a unique serial number is generated.

Input: Book details

Output: Unique serial number

R.1.2: Issue book

Description: A friend can be issued book only if he is registered. The various books outstanding against him along with the date borrowed are first displayed.

R.1.2.1: Display outstanding books

Description: First a friend's name and the serial number of the book to be issued are entered. Then the books outstanding against the friend should be displayed.

Input: Friend name

Output: List of outstanding books along with the date on which each was borrowed.

R.1.2.2: Confirm issue book

If the owner confirms, then the book should be issued to him and the relevant records should be updated.

Input: Owner confirmation for book issue. Output: Confirmation of book issue.

R.1.3: Query outstanding books

Description: Details of friends who have books outstanding against their name is displayed.

Input: User selection

Output: The display includes the name, address and telephone numbers of each friend against whom books are outstanding along with the titles of the outstanding books and the date on which those were issued.

R.1.4: Query book

Description: Any user should be able to query a particular book from anywhere using a web browser.

Input: Name of the book.

Output: Availability of the book and whether the book is issued out.

R.1.5: Return book

Description: Upon return of a book by a friend, the date of return is stored and the book is removed from the borrowing list of the concerned friend.

Input: Name of the book.

Output: Confirmation message.

2. Manage friend details

R.2.1: Register friend

Description: A friend must be registered before he can be issued books. After the registration data is entered correctly, the data should be stored and a confirmation message should be displayed.

Input: Friend details including name of the friend, address, land line number and mobile number.

Output: Confirmation of registration status.

R.2.2: Update friend details

Description: When a friend's registration information changes, the same must be updated in the computer.

R.2.2.1: Display current details

Input: Friend name.

Output: Currently stored details.

R2.2.2: Update friend details

Input: Changes needed.

Output: Updated details with confirmation of the changes.

R.3.3: Delete a friend record

Description: Delete records of inactive members.

Input: Friend name.

Output: Confirmation message.

3. Manage borrowed books

R.3.1: Register borrowed books

Description: The books borrowed by the user of the personal library are registered.

Input: Title of the book and the date borrowed.

Output: Confirmation of the registration status.

R.3.2: Deregister borrowed books

Description: A borrowed book is deregistered when it is returned.

Input: Book name.

Output: Confirmation of deregistration.

R.3.3: Display borrowed books

Description: The data about the books borrowed by the owner are displayed.

Input: User selection.

Output: List of books borrowed from other friends.

4. Manage statistics

R.4.1: Display book count

Description: The total number of books in the personal library should be displayed.

Input: User selection.

Output: Count of books.

R4.2: Display amount invested

Description: The total amount invested in the personal library is displayed.

Input: User selection.

Output: Total amount invested.

R.4.2: Display number of transactions *Description:* The total numbers of books issued and returned over a specific period by one (or all) friend(s) is displayed.

Input: Start of period and end of period.

Output: Total number of books issued and total number of books returned.

Non-functional requirements

N.1: Database: A data base management system that is available free of cost in the public domain should be used.

N.2: Platform: Both Windows and Unix versions of the software need to be developed. **N.3: Web-support:** It should be possible to invoke the query book functionality from any place by using a web browser.

Observation: Since there are many functional requirements, the requirements have been organised into four sections: Manage own books, manage friends, manage borrowed books, and manage statistics. Now each section has less than 7 functional requirements. This would not only enhance the readability of the document, but would also help in design.

4.2.11 Techniques for Representing Complex Logic

A good SRS document should properly characterise the conditions under which different scenarios of interaction occur (see Section 4.2.5). That is, a high-level function might involve different steps to be undertaken as a consequence of some decisions made after each step. Sometimes the conditions can be complex and numerous and several alternative interaction and processing sequences may exist depending on the outcome of the corresponding condition checking. A simple text description in such cases can be difficult to comprehend and analyse. In such situations, a decision tree or a decision table can be used to represent the logic and the processing involved. Also, when the decision making in a functional requirement has been represented as a decision table, it becomes easy to automatically or at least manually design test

cases for it. However, use of decision trees or tables would be superfluous in cases where the number of alternatives are few, or the decision logic is straightforward. In such cases, a simple text description would suffice.

There are two main techniques available to analyse and represent complex processing logic—decision trees and decision tables. Once the decision making logic is captured in the form of trees or tables, the test cases to validate these logic can be automatically obtained. It should, however, be noted that decision trees and decision tables have much broader applicability than just specifying complex processing logic in an SRS document. For instance, decision trees and decision tables find applications in information theory and switching theory.

Decision tree

A decision tree gives a graphic view of the processing logic involved in decision making and the corresponding actions taken. Decision tables specify which variables are to be tested, and based on this what actions are to be taken depending upon the outcome of the decision making logic, and the order in which decision making is performed.

The edges of a decision tree represent conditions and the leaf nodes represent the actions to be performed depending on the outcome of testing the conditions. Instead of discussing how to draw a decision tree for a given processing logic, we shall explain through a simple example how to represent the processing logic in the form of a decision tree.

Example 4.10 A library membership management software (LMS) should support the following three options—new member, renewal, and cancel membership. When the *new member* option is selected, the software should ask the member's name, address, and phone number. If proper information is entered, the software should create a membership record for the new member and print a bill for the annual membership charge and the security deposit payable. If the *renewal* option is chosen, the LMS should ask the member's name and his membership number and check whether he is a valid member. If the member details entered are valid, then the membership expiry date in the membership record should be updated and the annual membership charge payable by the member should be printed. If the membership details entered are invalid, an error message should be displayed. If the *cancel membership* option is selected and the name of a valid member is entered, then the membership is cancelled, a choke for the

balance amount due to the member is printed and his membership record is deleted. The decision tree representation for this problem is shown in Figure 4.3.

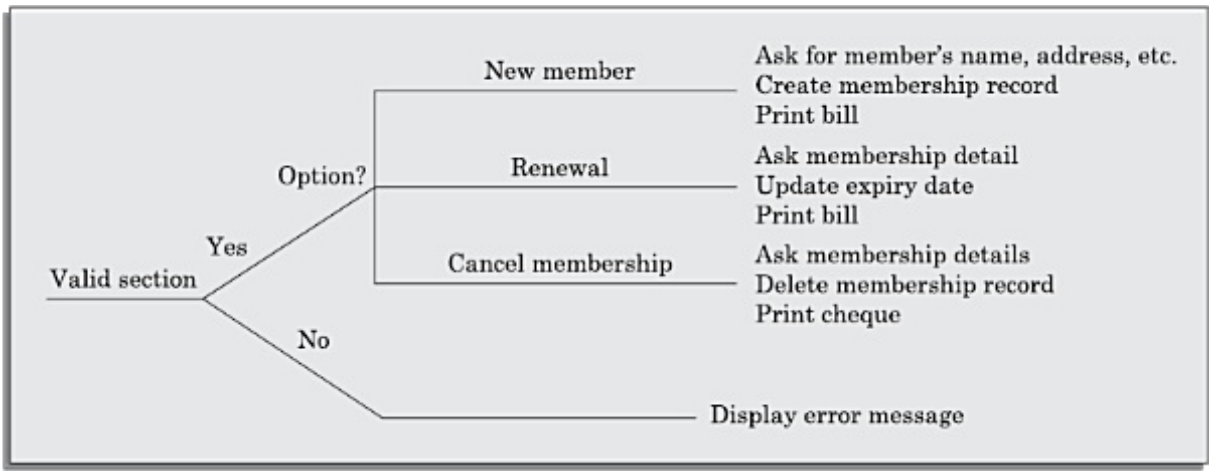

Figure 4.3: Decision Tree for LMS.

Observe from Figure 4.3 that the internal nodes represent conditions, the edges of the tree correspond to the outcome of the corresponding conditions. The leaf nodes represent the actions to be performed by the system. In the decision tree of Figure 4.3, first the user selection is checked. Based on whether the selection is valid, either further condition checking is undertaken or an error message is displayed. Observe that the order of condition checking is explicitly represented.

Decision table

A decision table shows the decision making logic and the corresponding actions taken in a tabular or a matrix form. The upper rows of the table specify the variables or conditions to be evaluated and the lower rows specify the actions to be taken when an evaluation test is satisfied. A column in the table is called a *rule*. A rule implies that if a certain condition combination is true, then the corresponding action is executed. The decision table for the

LMS problem of Example 4.10 is as shown in Table 4.1.

Table 4.1: Decision Table for the LMS Problem

Conditions				
Valid selection	NO	YES	YES	YES
New member	-	YES	NO	NO
Renewal	-	NO	YES	NO
Cancellation	-	NO	NO	YES
Actions				

Display error message	×			
Ask member's name, etc.		×		
Build customer record		×		
Generate bill		×	×	
Ask membership details			×	×
Update expiry date			×	
Print cheque				×
Delete record				×

Decision table *versus* decision tree

Even though both decision tables and decision trees can be used to represent complex program logic, they can be distinguishable on the following three considerations:

Readability: Decision trees are easier to read and understand when the number of conditions are small. On the other hand, a decision table causes the analyst to look at every possible combination of conditions which he might otherwise omit.

Explicit representation of the order of decision making: In contrast to the decision trees, the order of decision making is abstracted out in decision tables. A situation where decision tree is more useful is when multilevel decision making is required. Decision trees can more intuitively represent multilevel decision making hierarchically, whereas decision tables can only represent a single decision to select the appropriate action for execution.

Representing complex decision logic: Decision trees become very complex to understand when the number of conditions and actions increase. It may even be to draw the tree on a single page. When very large number of decisions are involved, the decision table representation may be preferred.

4.3 FORMAL SYSTEM SPECIFICATION

In recent years, formal techniques[3] have emerged as a central issue in software engineering. This is not accidental; the importance of precise specification, modelling, and verification is recognised to be important in most engineering disciplines. Formal methods provide us with tools to precisely describe a system and show that a system is correctly implemented. We say a system is correctly implemented when it satisfies its given specification. The specification of a system can be given either as a list of its desirable properties (property-oriented approach) or as an abstract model of the system (model-oriented

approach). These two approaches are discussed here. Before discussing representative examples of these two types of formal specification techniques, we first discuss a few basic concepts in formal specification We will first highlight some important concepts in formal methods, and examine the merits and demerits of using formal techniques.

4.3.1 What is a Formal Technique?

A formal technique is a mathematical method to specify a hardware and/or software system, verify whether a specification is realisable, verify that an implementation satisfies its specification, prove properties of a system without necessarily running the system, etc. The mathematical basis of a formal method is provided by its specification language. More precisely, a formal specification language consists of two sets—syn and sem, and a relation sat between them. The set *syn* is called the *syntactic domain*, the set *sem* is called the *semantic domain*, and the relation *sat* is called the *satisfaction relation*. For a given specification *syn*, and model of the system *sem*, if *sat (syn, sem)*, then *syn* is said to be the *specification of sem*, and *sem* is said to be the *specificand* of *syn*.

The generally accepted paradigm for system development is through a hierarchy of abstractions. Each stage in this hierarchy is an implementation of its preceding stage and a specification of the succeeding stage. The different stages in this system development activity are requirements specification, functional design, architectural design, detailed design, coding, implementation, etc. In general, formal techniques can be used at every stage of the system development activity to verify that the output of one stage conforms to the output of the previous stage.

Syntactic domains

The syntactic domain of a formal specification language consists of an alphabet of symbols and a set of formation rules to construct well-formed formulas from the alphabet. The well-formed formulas are used to specify a system.

Semantic domains

Formal techniques can have considerably different semantic domains. Abstract data type specification languages are used to specify algebras, theories, and programs. Programming languages are used to specify functions from input to output values. Concurrent and distributed

system specification languages are used to specify state sequences, event sequences, state-transition sequences, synchronisation trees, partial orders, state machines, etc.

Satisfaction relation

Given the model of a system, it is important to determine whether an element of the semantic domain satisfies the specifications. This satisfaction is determined by using a homomorphism known as *semantic abstraction function*. The semantic abstraction function maps the elements of the semantic domain into equivalent classes. There can be different specifications describing different aspects of a system model, possibly using different specification languages. Some of these specifications describe the system's behaviour and the others describe the system's structure. Consequently, two broad classes of semantic abstraction functions are defined— those that *preserve* a system's behaviour and those that *preserve* a system's structure.

Model *versus* property-oriented methods

Formal methods are usually classified into two broad categories—the so-called *model-oriented* and the property-oriented approaches. In a *model-oriented* style, one defines a system's behaviour directly by constructing a model of the system in terms of mathematical structures such as tuples, relations, functions, sets, sequences, etc. In the *property-oriented* style, the system's behaviour is defined indirectly by stating its properties, usually in the form of a set of axioms that the system must satisfy. Let us consider a simple producer/consumer example. In a *property-oriented* style, we would probably start by listing the properties of the system like—the consumer can start consuming only after the producer has produced an item, the producer starts to produce an item only after the consumer has consumed the last item, etc. Two examples of property-oriented specification styles are axiomatic specification and algebraic specification.

In *a model-oriented* style, we would start by defining the basic operations, p (produce) and c (consume). Then we can state that S 1 + p $\Rightarrow$ S, S + c $\Rightarrow$ S 1. Thus model-oriented approaches essentially specify a program by writing another, presumably simpler program. A few notable examples of popular model-oriented specification techniques are Z, CSP,CCS, etc.

It is alleged that property-oriented approaches are more suitable for *requirements specification*, and that the model-oriented approaches are more suited to *system design specification*. The reason for this distinction is the fact that property-oriented approaches specify a system behaviour not by what they say of the system but by what they do not say of the system. Thus, property-oriented specifications permit a large number of possible implementations. Furthermore, property-oriented approaches specify a system by a conjunction of axioms, thereby making it easier to alter/augment specifications at a later stage. On the other hand, model-oriented methods do not support logical conjunctions and disjunctions, and thus even minor changes to a specification may lead to overhauling an entire specification. Since the initial customer requirements undergo several changes as the development proceeds, the property-oriented style is generally preferred for requirements specification. Later in this chapter, we have discussed two property-oriented specification techniques.

4.3.2 Operational Semantics

Informally, the *operational semantics* of a formal method is the way computations are represented. There are different types of operational semantics according to what is meant by a *single run* of the system and how the runs are grouped together to describe the *behaviour* of the system. In the following subsection we discuss some of the commonly used operational semantics.

Linear semantics: In this approach, a *run* of a system is described by a sequence (possibly infinite) of events or states. The concurrent activities of the system are represented by non-deterministic interleavings of the atomic actions. For example, a concurrent activity a || b is represented by the set of sequential activities a; b and b; a. This is a simple but rather unnatural representation of concurrency. The behaviour of a system in this model consists of the set of all its runs. To make this model more realistic, usually *justice* and *fairness* restrictions are imposed on computations to exclude the unwanted interleavings.

Branching semantics: In this approach, the behaviour of a system is represented by a directed graph. The nodes of the graph represent the possible states in the evolution of a system. The descendants of each node of the graph represent the states which can be generated by any of the atomic actions enabled at that state. Although this semantic model distinguishes the

branching points in a computation, still it represents concurrency by interleaving.

Maximally parallel semantics: In this approach, all the concurrent actions enabled at any state are assumed to be taken together. This is again not a natural model of concurrency since it implicitly assumes the availability of all the required computational resources.

Partial order semantics: Under this view, the semantics ascribed to a system is a *structure of states* satisfying a partial order relation among the states (events). The partial order represents a *precedence ordering* among events, and constrains some events to occur only after some other events have occurred; while the occurrence of other events (called *concurrent events*) is considered to be incomparable. This fact identifies concurrency as a phenomenon not translatable to any interleaved representation.

Merits and limitations of formal methods

In addition to facilitating precise formulation of specifications, formal methods possess several positive features, some of which are discussed as follows:

- Formal specifications encourage rigour. It is often the case that the very process of construction of a rigorous specification is more important than the formal specification itself. The construction of a rigorous specification clarifies several aspects of system behaviour that are not obvious in an informal specification. It is widely acknowledged that it is cost-effective to spend more efforts at the specification stage, otherwise, many flaws would go unnoticed only to be detected at the later stages of software development that would lead to iterative changes to occur in the development life cycle. According to an estimate, for large and complex systems like distributed real-time systems 80 per cent of project costs and most of the cost overruns result from the iterative changes required in a system development process due to inappropriate formulation of requirements specification. Thus, the additional effort required to construct a rigorous specification is well worth the trouble.
- Formal methods usually have a well-founded mathematical basis. Thus, formal specifications are not only more precise, but also mathematically sound and can be used to reason about the properties

of a specification and to rigorously prove that an implementation satisfies its specifications. Informal specifications may be useful in understanding a system and its documentation, but they cannot serve as a basis of verification. Even carefully written specifications are prone to error, and experience has shown that unverified specifications are comparable in reliability to unverified programs. automatically avoided when one formally specifies a system.

- The mathematical basis of the formal methods makes it possible for automating the analysis of specifications. For example, a tableau-based technique has been used to automatically check the consistency of specifications. Also, automatic theorem proving techniques can be used to verify that an implementation satisfies its specifications. The possibility of automatic verification is one of the most important advantages of formal methods.
- Formal specifications can be executed to obtain immediate feedback on the features of the specified system. This concept of executable specifications is related to rapid prototyping. Informally, a prototype is a "toy" working model of a system that can provide immediate feedback on the behaviour of the specified system, and is especially useful in checking the completeness of specifications.

It is clear that formal methods provide mathematically sound frameworks within which large, complex systems can be specified, developed and verified in a systematic rather than in an ad hoc manner. However, formal meth ods suffer from several shortcomings, some of which are as following:

- Formal methods are difficult to learn and use.
- The basic incompleteness results of first-order logic suggest that it is impossible to check *absolute* correctness of systems using theorem proving techniques.
- Formal techniques are not able to handle complex problems. This shortcoming results from the fact that, even moderately complicated problems blow up the complexity of formal specification and their analysis. Also, a large unstructured set of mathematical formulas is difficult to comprehend.

It has been pointed out by several researchers that formal specifications neither replace nor make the informal descriptions obsolete but complement

them. In fact, the comprehensibility of formal specifications is greatly enhanced when the specifications are accompanied by an informal description. What is suggested is the use of formal techniques as a broad guideline for the use of the informal techniques. An interesting example of such an approach is reported by Jones in [1980]. In this approach, the use of a formal method identifies the necessary verification steps that need to be carried out, but it is legitimate to apply informal reasoning in presentation of correctness arguments and transformations. Any doubt or query relating to an informal argument is to be resolved by formal proofs.

In the following two sections, we discuss the axiomatic and algebraic specification styles. Both these techniques can be classified as the property-oriented specification techniques.

4.4 AXIOMATIC SPECIFICATION

In axiomatic specification of a system, first-order logic is used to write the pre- and post- conditions to specify the operations of the system in the form of axioms. The pre-conditions basically capture the conditions that must be satisfied before an operation can successfully be invoked. In essence, the pre-conditions capture the requirements on the input parameters of a function. The post-conditions are the conditions that must be satisfied when a function post-conditions are essentially constraints on the results produced for the function execution to be considered successful.

How to develop an axiomatic specifications?

The following are the sequence of steps that can be followed to systematically develop the axiomatic specifications of a function:

- Establish the range of input values over which the function should behave correctly. Establish the constraints on the input parameters as a predicate.
- Specify a predicate defining the condition which must hold on the output of the function if it behaved properly.
- Establish the changes made to the function's input parameters after execution of the function. Pure mathematical functions do not change their input and therefore this type assertion is not necessary for pure functions.
- Combine all of the above into pre- and post-conditions of the function.

We now illustrate how simple abstract data types can be algebraically specified through two simple examples.

Example 4.11 Specify the pre- and post-conditions of a function that takes a real number as argument and returns half the input value if the input is less than or equal to 100, or else returns double the value.

$$f\ (x : real) : real$$

$$pre : x \in R$$

$$post : \{(x \leq 100) \wedge (f\ (x) = x/2)\} \vee \{(x > 100) \wedge (f\ (x) = 2 * x)\}$$

Example 4.12 Axiomatically specify a function named search which takes an integer array and an integer key value as its arguments and returns the index in the array where the key value is present.

$$search(X : intArray, key : integer) : integer$$

$$pre : \exists i \in [X\ first...X\ last], X\ [i] = key$$

$$post : \{(X' [search(X, key)] = key) \wedge (X = X')\}$$

Please note that we have followed the convention that if a function changes any of its input parameters, and if that parameter is named X , then we refer to it after the function completes execution as X ’. One practical application of the axiomatic specification is in program documentation. Engineers developing code for a function specify the function by noting down the pre and post conditions of the function in the function header. Another application of the axiomatic specifications is in proving program properties by composing the pre and post-conditions of a number of functions.

4.5 ALGEBRAIC SPECIFICATION

In the algebraic specification technique, an object class or type is specified in terms of relationships existing between the operations defined on that type. It was first brought into prominence by Guttag [1980,1985] in specification of abstract data types. Various notations of algebraic specifications have evolved, including those based on OBJ and Larch languages.

Essentially, algebraic specifications define a system as a *heterogeneous algebra*. A heterogeneous algebra is a collection of different sets on which several operations are defined. Traditional algebras are homogeneous. A homogeneous algebra consists of a single set and several operations defined in this set; e.g. { I, +, -, *, / }. In contrast, alphabetic strings S together with

operations of concatenation and length {S, I, con, len}, is not a homogeneous algebra, since the range of the length operation is the set of integers.

Each set of symbols in a heterogeneous algebra is called a *sort* of the algebra. To define a heterogeneous algebra, besides defining the sorts, we need to specify the involved operations, their signatures, and their domains and ranges. Using algebraic specification, we define the meaning of a set of interface procedure by using *equations*. An algebraic specification is usually presented in four sections.

Types section: In this section, the sorts (or the data types) being used is specified.

Exception section: This section gives the names of the exceptional conditions that might occur when different operations are carried out. These exception conditions are used in the later sections of an algebraic specification.

Syntax section: This section defines the signatures of the interface procedures. The collection of sets that form input domain of an operator and the sort where the output is produced are called the *signature* of the operator. For example, PUSH takes a stack and an element as its input and returns a new stack that has been created.

Equations section: This section gives a set of *rewrite rules* (or equations) defining the meaning of the interface procedures in terms of each other. In general, this section is allowed to contain conditional expressions.

By convention each equation is implicitly universally quantified over all possible values of the variables. This means that the equation holds for all possible values of the variable. Names not mentioned in the syntax section such r or e are variables. The first step in defining an algebraic specification is to identify the set of required operations. After having identified the required operators, it is helpful to classify them as either basic constructor operators, extra constructor operators, basic inspector operators, or extra inspection operators. The definition of these categories of operators is as follows:

Basic construction operators: These operators are used to create or modify entities of a type. The basic construction operators are essential to generate all possible element of the type being specified. For example, 'create' and 'append' are basic construction operators in Example 4.13.

Extra construction operators: These are the construction operators other

than the basic construction operators. For example, the operator 'remove' in Example 4.13 is an extra construction operator, because even without using 'remove' it is possible to generate all values of the type being specified.

Basic inspection operators: These operators evaluate attributes of a type without modifying them, e.g., eval, get, etc. Let S be the set of operators whose range is not the data type being specified—these are the inspection operators. The set of the basic operators S1 is a subset of S , such that each operator from S -S 1 can be expressed in terms of the operators from S 1.

Extra inspection operators: These are the inspection operators that are not basic inspectors. A simple way to determine whether an operator is a constructor (basic or extra) or an inspector (basic or extra) is to check the syntax expression for the operator. If the type being specified appears on the right hand side of the expression then it is a constructor, otherwise it is an inspection operator. For example, in Example 4.13, create is a constructor because point appears on the right hand side of the expression and point is the data type being specified. But, xcoord is an inspection operator since it does not modify the point type.

A good rule of thumb while writing an algebraic specification, is to first establish which are the constructor (basic and extra) and inspection operators (basic and extra). Then write down an axiom for composition of each basic construction operator over each basic inspection operator and extra constructor operator. Also, write down an axiom for each of the extra inspector in terms of any of the basic inspectors. Thus, if there are m_1 basic constructors, m_2 extra constructors, n_1 basic inspectors, and n_2 extra inspectors, we should have $m_1 \times (m_2 + n_1) + n_2$ axioms. However, it should be clearly noted that these $m_1 \times (m_2 + n_1) + n_2$ axioms are the minimum required and many more axioms may be needed to make the specification complete. Using a complete set of rewrite rules, it is possible to simplify an arbitrary sequence of operations on the interface procedures.

While developing the rewrite rules, different persons can come up with different sets of equations. However, while developing the equations one has to be careful that the equations should be able to handle all meaningful composition of operators, and they should have the unique termination and finite termination properties. These two properties of the rewrite rules are discussed later in this section.

Example 4.13 Let us specify a data type point supporting the operations create, xcoord, ycoord, isequal; where the operations have their usual

meaning.

```
Types:
   defines point
   uses boolean, integer
```

Syntax:

1. create : integer × integer → point
2. xcoord : point → integer
3. ycoord : point → integer
4. isequal : point × point → boolean

Equations:

1. xcoord(create(x, y)) = x
2. ycoord(create(x, y)) = y
3. isequal(create(x1, y1), create(x2, y2)) = ((x1 = x2)and(y1 = y2))

In this example, we have only one basic constructor (create), and three basic inspectors (xcoord, ycoord, and isequal). Therefore, we have only 3 equations.

The rewrite rules let you determine the meaning of any sequence of calls on the point type. Consider the following expression: *isequal (create (xcoord (create(2, 3)), 5),create (ycoord (create(2, 3)), 5))*. By applying the rewrite rule 1, you can simplify the given expression as *isequal (create (2, 5), create (ycoord (create(2, 3)), 5))*. By using rewrite rule 2, you can further simplify this as *isequal (create (2, 5),create (3, 5))*. This is false by rewrite rule 3.

Properties of algebraic specifications

Three important properties that every algebraic specification should possess are:

Completeness: This property ensures that using the equations, it should be possible to reduce any arbitrary sequence of operations on the interface procedures. When the equations are not complete, at some step during the reduction process, we might not be able to reduce the expression arrived at that step by using any of the equations. There is no simple procedure to ensure that an algebraic specification is complete.

Finite termination property: This property essentially addresses the following question: Do applications of the rewrite rules to arbitrary expressions involving the interface procedures always terminate? For arbitrary algebraic equations, convergence (finite termination) is undecidable.

But, if the right hand side of each rewrite rule has fewer terms than the left, then the rewrite process must terminate.

Unique termination property: This property indicates whether application of rewrite rules in different orders always result in the same answer. Essentially, to determine this property, the answer to the following question needs to be checked—Can all possible sequence of choices in application of the rewrite rules to an arbitrary expression involving the interface procedures always give the same answer? Checking the unique termination property is a very difficult problem.

Example 4.14 Let us specify a FIFO queue supporting the operations create, append, remove, first, and isempty; where the operations have their usual meaning.

```
Types:
    defines queue
    uses boolean, element
Exception:
    underflow, novalue
Syntax:
```

1. create : $\phi \rightarrow$ queue
2. append : queue $\times$ element $\rightarrow$ queue
3. remove : queue $\rightarrow$ queue + {underf low}
4. f irst : queue $\rightarrow$ element + {novalue}
5. isempty : queue $\rightarrow$ boolean

```
Equations:
```

1. isempty(create()) = true
2. isempty(append(q, e)) = f alse
3. f irst(create()) = novalue
4. f irst(append(q, e)) = if isempty(q) then e else f irst(q)
5. remove(create()) = underf low
6. remove(append(q, e)) = if isempty(q) then create( ) else append(remove(q), e)

In this example, we have two basic construction operators (create and append). We have one extra construction operator (remove). We have considered remove to be an extra construction operator because all values of the queue can be realised, even without having the remove operator. We have two basic inspectors (first and isempty). Therefore we have $2 \times 3 = 6$

equations.

4.5.1 Auxiliary Functions

Sometimes while specifying a system, one needs to introduce extra functions not part of the system to define the meaning of some interface procedures. These are called *auxiliary functions*. In the following, we discuss an example where it becomes necessary to use an auxiliary function to be able to specify a system.

Example 4.15 Let us specify a bounded FIFO queue having a maximum size of MaxSize and supporting the operations create, append, remove, first, and isempty; where the operations have their usual meaning.

```
Types:
    defines queue
    uses boolean, element, integer
Exception:
    underflow, novalue, overflow
Syntax:
```

1. create : $\phi \rightarrow$ queue
2. append : queue $\times$ element $\rightarrow$ queue + {overf low}
3. size : queue $\rightarrow$ integer
4. remove : queue $\rightarrow$ queue + {underf low}
5. f irst : queue $\rightarrow$ element + {novalue}
6. isempty : queue $\rightarrow$ boolean

```
Equations:
```

1. f irst(create()) = novalue
2. *first(append(q,e)) = if size(q)= MaxSize then overflow else if isempty(q) then e else first(q)*
3. remove(create()) = underf low
4. remove(append(q, e)) = if isempty(q) then create() else
   .
   if size(q) = M axS ize then overf low else append(remove(q), e)
5. size(create()) = 0
6. size(append(q, e)) = if size(q) = M axS ize then overf low else size(q) + 1
7. isempty(q) = if (size(q) = 0) then true elsef alse

In this example, we have used the auxiliary function size to enable us to

specify that during appending an element, overflow might occur if the queue size exceeds MaxSize. However, after we have introduced the auxiliary function size, we find that the operator isempty can no longer be considered as a basic inspector because isempty can be expressed in terms of size. Therefore, we have removed the axioms for the operator issmpty used in Example 4.15, and have instead used an axiom to express isempty in terms of size. We have added two axioms to express size in terms of the basic construction operators (create and append).

4.5.2 Structured Specification

Developing algebraic specifications is time consuming. Therefore efforts have been made to devise ways to ease the task of developing algebraic specifications. The following are some of the techniques that have successfully been used to reduce the effort in writing the specifications.

Incremental specification: The idea behind incremental specification is to first develop the specifications of the simple types and then specify more complex types by using the specifications of the simple types.

Specification instantiation: This involves taking an existing specification which has been developed using a generic parameter and instantiating it with some other sort.

Pros and Cons of algebraic specifications

Algebraic specifications have a strong mathematical basis and can be viewed as heterogeneous algebra. Therefore, they are unambiguous and precise. Using an algebraic specification, the effect of any arbitrary sequence of operations involving the interface procedures can automatically be studied. A major shortcoming of algebraic specifications is that they cannot deal with side effects. Therefore, algebraic specifications are difficult to integrate with typical programming languages. Also, algebraic specifications are hard to understand.

4.6 EXECUTABLE SPECIFICATION AND 4GL

When the specification of a system is expressed formally or is described by using a programming language, then it becomes possible to directly execute the specification without having to design and write code for

implementation. However, executable specifications are usually slow and inefficient, 4GLs[4] (4th Generation Languages) are examples of executable specification languages. 4GLs are successful because there is a lot of large granularity commonality across data processing applications which have been identified and mapped to program code. 4GLs get their power from software reuse, where the common abstractions have been identified and parameterized. Careful experiments have shown that rewriting 4GL programs in 3GLs results in up to 50 per cent lower memory usage and also the program execution time can reduce up to ten folds.

SUMMARY

- Substantial time and effort must be spent in developing good quality SRS document before starting the design activity. Any improper specification turns out to be very expensive as the results of this phase affect all subsequent phases of development.
- The requirements analysis and specification phase consists of two important activities—requirements gathering and analysis, and requirements specification.
- The aim of requirements analysis is to clearly understand the exact user requirements and to remove any inconsistencies, anomalies, and incompleteness in these requirements.
- During the requirements specification activity, the requirements are systematically organised into an SRS document.
- Formally specifying the requirements has many advantages. But, a major shortcoming of the formal specification techniques is that they are hard to use. However, it is possible that formal techniques will become more usable in future with the development of suitable front-ends. We discussed the axiomatic and algebraic techniques as example formal specification techniques to give an idea of some of the issues involved in formal specification.

EXERCISES

1. Choose the correct option.
   (a) Who among the following is a stakeholder in a software development project?
   (i) A shareholder of the organisation developing the software

(ii) Anyone who is interested in the software
(iii) Anyone who is a source of requirements for the software
(iv) Anyone who might be affected by the software

(b) A software requirements specification (SRS) document should avoid discussing which one of the following?
(i) Functional requirements
(ii) Non-functional requirements
(iii) Design specification
(iv) Constraints on the implementation

(c) Which of the following is not a goal of requirements analysis?
(i) Weed out ambiguities in the requirements
(ii) Weed out inconsistencies in the requirements
(iii) Weed out non-functional requirements
(iv) Weed out incompleteness in the requirements

(d) Consider the following requirement for a word processor software: "The software should provide facility to import an existing image available as a jpeg file into the document being created." Which one of the following types of requirement is this?
(i) Functional requirement
(ii) Non-functional requirement
(iii) Constraint on the implementation
(iv) Goal of implementation

(e) Assume that you are the project manager of a development project for a data processing application in which the user requirements for the GUI part are not very clear. Which life cycle model would you use to develop the GUI part?
(i) Classical waterfall model (ii) Iterative waterfall model (iii) Prototyping model
(iv) Spiral model

2. What is the difference between requirements analysis and specification? What are the important activities carried out during requirements analysis and specification phase? What is the final outcome of the requirements analysis and specification phase?
3. What are the goals of the requirements analysis and specification phase? How are the requirements analysis and specification activities carried out and by whom?
4. Discuss the important ways in which a well formulated SRS document can be useful to various stakeholders.

5. What is the difference between the functional and the non-functional requirements of a system? Identify at least two functional requirements of a bank automated teller machine (ATM) system. Also identify one non-functional requirement for an ATM system.
6. What are the four types of non-functional requirements that have been suggested by IEEE 830 standard document. Give one example of each of these categories of requirements.
7. What do you understand by requirements gathering? Name and explain the different requirements gathering techniques that are normally deployed by an analyst.
8. What are the different types of requirements problems that an analyst usually anticipates and rectifies in the gathered requirements before starting to write the SRS document? Give at least one example of each.
9. Explain the likely consequences of starting a large project development effort without accurately understanding and documenting the customer requirements.
10. Suppose you have been appointed as the analyst for a large software development project. Discuss the aspects of the software product you would document in the software requirements specification (SRS) document? What would be the organisation of your SRS document? How would you validate your SRS document?
11. Make a checklist of errors that might exist in an SRS document. This checklist can be used to review an SRS document.
12. Write down the important users of the SRS document for a project, the specific ways in which they use the document, and their specific expectations from the document, if any.
13. What do you understand by the problems of overspecification, forward reference, and noise in an SRS document? Explain each of these with suitable examples.
14. What is the difference between functional and non-functional requirements? Give one example of each type of requirement for a library automation software.
15. List five desirable characteristics of a good software requirements specification (SRS) document.
16. Suppose you are trying to gather the requirements for a software that needs to be developed to automate the book-keeping activities of a supermarket. Identify the main tasks that you would undertake as the analyst to satisfactorily gather the requirements.

17. How are the abstraction and decomposition principles used in the development of a good software requirements specification?
18. Suppose the analyst of a large product development effort has prepared the SRS document in the form of a narrative essay of the system to be developed. Based on this document, the product development activity gets underway. Explain the problems that such a requirements specification document may create while developing the software.
19. Discuss the relative advantages of formal and informal requirements specifications.
20. Why is the SRS document also known as the black-box specification of a system?
21. Who are the different category of users of the SRS document? In what ways is the SRS document useful to them?
22. Give an example of an inconsistent functional requirement. Explain why do you think that the requirement is inconsistent.
23. What do you understand by traceability in the context of software requirements specification. How is traceability achieved? Identify at least two important benefits of having traceability among development artifacts.
24. State whether the following statements are **TRUE** o r **FALSE**. Give reasons for your answer.
    (a) Applications developed using 4GLs would normally be more efficient and run faster compared to applications developed using 3GL. (b) A formal specification cannot be ambiguous.
    (c) A formal specification cannot be incomplete. (d) A formal specification cannot be inconsistent.
    (e) The system test plan can be prepared immediately after the completion of the requirements specification phase.
    (f) The SRS document is a formal specification of a system.
    (g) User interface issues of a system are usually its functional requirements.
    (h) The SRS document is written using the customer's terminology of various data and procedures in the problem, rather than the development team's terminology.
    (i) A precise specification cannot be incomplete.
    (j) If a requirement specification is precise, then it would automatically imply that it is an unambiguous requirements specification.

25. (a) What are the important differences between a model-oriented specification method and a property-oriented specification method.
    (b) Compare the relative advantages of property-oriented specification methods over model-oriented specification methods.
    (c) Name at least one representative popular property-oriented specification technique, and one representative model-oriented specification technique.
26. Consider the following requirement for a software to be developed for controlling a chemical plant. The chemical plant has a number of emergency conditions. When any of the emergency conditions occurs, some prespecified actions should be taken. The different emergency conditions and the corresponding actions that need to be taken are as follows:

(a) If the temperature of the chemical plant exceeds T_1 □C, then the water shower should be turned ON and the heater should be turned OFF.
    (b) If the temperature of the chemical tank falls below T2 □ C, then the heater should be turned ON and the water shower should be turned OFF.
    (c) If the pressure of the chemical plant is above P_1, then the valve v_1 should be OPENED.
    (d) If the chemical concentration of the tank rises above M , and the temperature of the tank is more than T3 □ C, then the water shower should be turned ON.
    (e) If the pressure rises above P_3 and the temperature rises above T_1 □ C, then the water shower should be turned ON, valves v_1 and v_2 are OPENED and the alarm bells sounded.

    Write the requirements of this chemical plant software in the form of a decision table.
27. Draw a decision tree to represent the processing logic of the chemical plant controller described in question 26.
28. Represent the decision making involved in the operation of the following wash-machine by means of a decision table:
    The machine waits for the `start` switch to be pressed. After the user presses the `start` switch, the machine fills the wash tub with either hot or cold water depending upon the setting of the `HotWash` switch. The water filling continues until the high level is sensed. The machine starts the agitation motor and continues agitating the wash tub until either the

preset timer expires or the user presses the stop switch. After the agitation stops, the machine waits for the user to press the startDrying switch. After the user presses the startDrying switch, the machine starts the hot air blower and continues blowing hot air into the drying chamber until either the user presses the stop switch or the preset timer expires.

29. Represent the processing logic of the following problem in the form of a decision table: A Library Membership Automation System needs to support three functions: **add new-member, renew-membership, cancel-membership**. If the user requests for any function other than these three, then an error message is is flashed. When an **add new-member** request is made, a new member record is created and a bill for the annual membership fee for the new member is generated. If a membership renewal request is made, then the expiry date of the concerned membership record is updated and a bill towards the annual membership fee is generated. If a membership cancellation request is made, then the concerned membership record is deleted and a cheque for the balance amount due to the member is printed.
30. What do you understand by pre- and post-conditions of a function? Write the pre- and post-conditions to axiomatically specify the following functions:
    (a) A function takes two floating point numbers representing the sides of a rectangle as input and returns the area of the corresponding rectangle as output.
    (b) A function accepts three integers in the range of -100 and +100 and determines the largest of the three integers.
    (c) A function takes an array of integers as input and finds the minimum value.
    (d) A function named square-array creates a 10 element array where each all elements of the array, the value of any array element is square of its index.
    (e) A function sort takes an integer array as its argument and sorts the input array in ascending order.
31. Using the algebraic specification method, formally specify a **string** supporting the following operations:
    - **append:** append a given string to another string
    - **add:** add a character to a string
    - **create:** create a new null string

- **isequal:** checks whether two strings are equal or not
- **isempty:** checks whether the string is null

32. Using the algebraic specification method, formally specify an **array** of generic type elem.
    Assume that array supports the following operations:
    - **create:** takes the array bounds as parameters and initializes the values of the array to undefined.
    - **eval:** reveals the value of a specified element.
    - **first:** returns the first bound of the array.
    - **last:** returns the last bound of the array.
33. What do you understand by an executable specification language? How is it different from a traditional procedural programming language? Give an example of an executable specification language.
34. What is a fourth generation programming technique? What are its advantages and disadvantages vis-a-vis a third generation technique?
35. What are auxiliary functions in algebraic specifications? Why are these needed?
36. What do you understand by incremental development of algebraic specifications? What is the advantage of incremental development of algebraic specifications?
37. (a) Algebraically specify an abstract data type that stores a **set** of elements and supports the following operations. Assume that the ADT element has already been specified and you can use it:
    - **new:** creates a null set.
    - **add:** takes a set and an element and returns the set with the additional elements stored.
    - **size:** takes a set as argument and returns the number of elements in the set.
    - **remove:** takes a set and an element as its argument and returns the set with the element removed.
    - **contains:** takes a set and an element as its argument and returns the boolean value true if the element belongs to the set and returns false if the element does not belong to the set.
    - **equals:** takes two sets as arguments and returns true if they contain identical elements and returns false otherwise.

    (b) Using the specification you have developed for the ADT set, reduce the following expression by applying the rewrite rules: equals (add(5,

(add(6, new()), add(6, (add(5, new())))). Show the details of every reduction.

38. Algebraically specify a data type Point, that supports the following operations: create, xcoord, ycoord, move, movex, movey. The informal meanings of these operations are the following—create takes two integers as its arguments and creates an instance of point type that has the two integers as its x and y coordinate values respectively, xcoord and ycoord return the x and y-coordinates of a given point, move takes a point and two integer values as its argument and sets the x and y-coordinates of point to the specified values, movex takes a point and an integer value as its argument and sets the x-coordinate of the point to the given integer value. Similarly, movey takes a point and an integer value and sets the y-coordinate of point to the given integer value.
Reduce the following expression, clearly showing each step and mentioning the reduction rule used.
xcoord(movex(create(20,100), ycoord(create(10,50)))

39. Write a formal algebraic specification of the sort **symbol-table** whose operations are informally defined as follows:
- **create:** bring a symbol table into existence.
- **enter:** enter a symbol table and its type into the table.
- **lookup:** return the type associated with a name in the table.
- **delete:** remove a name, type pair from the table, given a name as a parameter.
- **replace:** replace the type associated with a given name with the type specified as its parameter.

The enter operation fails if the name is already present in the table. The lookup, delete, and replace operations fail if the name is not available in the table.

40. Algebraically specify the data type **queue** which supports the following operations:
- **create:** creates an empty queue.
- **append:** takes a queue and an item as its arguments and returns a queue with the item added at the end of the queue.
- **remove:** takes a queue as its argument and returns a queue with the first element of the original queue removed.
- **inspect:** takes a queue as its argument and returns the value of the first item in the queue.

- **isempty:** takes a queue as its argument and returns true if the queue contains no elements, and returns false if it contains one or more elements.

You can assume that the data type item has previously been specified and that you can reuse this specification.

41. Define the finite termination and unique termination properties of algebraic specifications? Why is it necessary for an algebraic specification to satisfy these properties?
42. If the prototyping model is being used in a development effort, is it necessary to develop a requirements specification document?
43. Express the decision making involved in the following withdraw cash function of a bank ATM using a decision table.

    To withdraw cash, first a valid customer identification is required. For this, the customer is prompted to insert his ATM card in the card checking machine. If his card is found to be invalid, the card is ejected out along with an appropriate message displayed. If the card is verified to be a valid card, the customer is prompted to type his password. If the password is invalid, an error message is shown and the customer is prompted to enter his password again. If the customer enters incorrect password consecutively for three times, then his card is seized and he is asked to contact the bank manager. On the other hand, if the customer enters his password correctly, then he is considered to have validly identified himself and is prompted to enter the amount he needs to withdraw. If he enters an amount that is not a multiple of Rs. 100, he is prompted to enter the amount again. After he enters an amount that is a multiple of Rs. 100, the cash is dispensed if sufficient amount is available in his account and his card is ejected; otherwise his card is ejected out without any cash being dispensed along with a message display regarding insufficient fund position in his account.
44. Identify the functional and non-functional requirements in the following problem description and document them.

    A cosmopolitan clock software is to be developed that displays up to 6 clocks with the names of the city and their local times. The clocks should be aesthetically designed. The software should allow the user to change name of any city and change the time readings of any clock by typing typing c (for configure) on any clock. The user should also be able to toggle between a digital clock and an analog clock display by typing either d (for digital) or a (for analog) on a clock display. After the stand-

alone implementation works, a web-version should be developed that can be downloaded on a browser as an applet and run. The clock should use only the idle cycles on the computer it runs.

45. What do you understand by inconsistencies, anomalies, and incompletenesses in an SRS document. Identify the inconsistencies, anomalies, and incompletenesses in the following requirements of an academic activity automation software of an educational institute:
"The semester performance of each student is computed as the average academic performance for the semester. The guardians of all students having poor performance record in the semester are mailed a letter informing about the poor performance of the ward and intimating that repetition of poor performance in the subsequent semester can lead to expulsion. The extracurricular activities of a student are also graded and taken into consideration for determination of the semester performance."

46. Identify any inconsistencies, anomalies, and incompleteness that are present in the following requirements that were gathered by interviewing the clerks of the CSE department for developing an academic automation software (AAS): "The CGPA of each student is computed as the average performance for the semester. The parents of all students having poor performance are mailed a letter informing about the poor performance of their ward and with a request to convey a warning to the student that the poor performance should not be repeated."

47. Represent the decision making involved in the following functional requirement of a library automation system: **`Issue Item:`** An item when submitted at the counter along with the library identity card, first it is determined if the member has exceeded his quota. If he has exceeded his quota, then no items can be issued to him. If the requested item is a journal, then it is issued for two days only. If it is a book, then it is checked whether it is a reference book. Reference books can not be issued out. If it is not a reference book, it is determined if any one has reserved it. Reserved books can not be issued out. If the book issue request of the member meets all the mentioned criteria, then the book is issued to the member for one month, appropriate entry is made in the member's account and an issue slip is printed.

48. Suppose you wish to develop a word processing software that would have features similar to Microsoft Word. Develop the SRS document for

this word processing software.

1 Note that the customer and the users of a software may, in general, be different. For example, the customer may be an organisation and the users may be a few select employees of the organisation.

2 A safety-critical system is one whose improper working can result in financial loss, loss of property, or life.

3 Sections 4.3–4.5 can be omitted in a first level course on software engineering.

4 Programming languages are generally classified into four generations. The first generation (1GL) programming languages consist of machine language programs. The second generation (2GL) started when the assembly language was introduced. All procedural languages are classified as 3GLs. In procedural languages, in order to solve a problem, you would have to precisely write down "how" the required result can be obtained. This requires writing the exact procedures or the algorithmic steps that need to be followed to arrive at the result. In contrast, using a 4GL only the "what" parts have to be specified.

Chapter 5

SOFTWARE DESIGN

During the software design phase, the design document is produced, based on the customer requirements as documented in the SRS document. We can state the main objectives of the design phase, in other words, as follows.

> The activities carried out during the design phase (called as design process) transform the SRS document into the design document.

This view of a design process has been shown schematically in Figure 5.1. As shown in Figure 5.1, the design process starts using the SRS document and completes with the production of the design document. The design document produced at the end of the design phase should be implementable using a programming language in the subsequent (coding) phase.

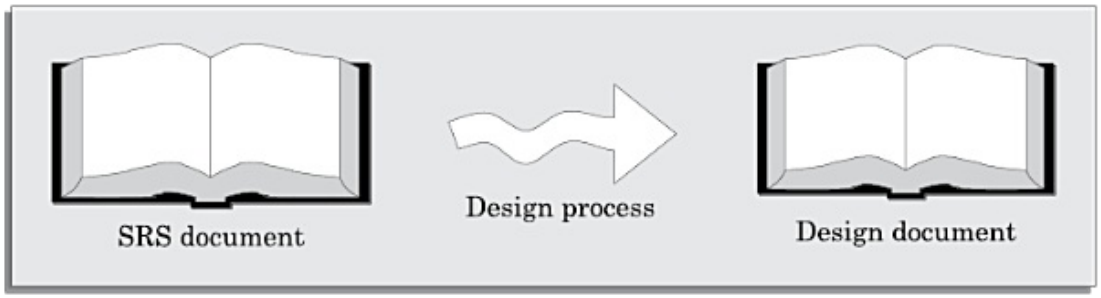

Figure 5.1: The design process.

5.1 OVERVIEW OF THE DESIGN PROCESS

The design process essentially transforms the SRS document into a design document. In the following sections and subsections, we will discuss a few important issues associated with the design process.

5.1.1 Outcome of the Design Process

The following items are designed and documented during the design phase.

Different modules required: The different modules in the solution should be clearly identified. Each module is a collection of functions and the data shared by the functions of the module. Each module should accomplish some well-defined task out of the overall responsibility of the software. Each module should be named according to the task it performs. For example, in an academic automation software, the module consisting of the functions and data necessary to accomplish the task of registration of the students should be named handle student registration.

Control relationships among modules: A control relationship between two modules essentially arises due to function calls across the two modules. The control relationships existing among various modules should be identified in the design document.

Interfaces among different modules: The interfaces between two modules identifies the exact data items that are exchanged between the two modules when one module invokes a function of the other module.

Data structures of the individual modules: Each module normally stores some data that the functions of the module need to share to accomplish the overall responsibility of the module. Suitable data structures for storing and managing the data of a module need to be properly designed and documented.

Algorithms required to implement the individual modules: Each function in a module usually performs some processing activity. The algorithms required to accomplish the processing activities of various modules need to be carefully designed and documented with due considerations given to the accuracy of the results, space and time complexities.

Starting with the SRS document (as shown in Figure 5.1), the design documents are produced through iterations over a series of steps that we are going to discuss in this chapter and the subsequent three chapters. The design documents are reviewed by the members of the development team to ensure that the design solution conforms to the requirements specification.

5.1.2 Classification of Design Activities

A good software design is seldom realised by using a single step procedure, rather it requires iterating over a series of steps called the design activities. Let us first classify the design activities before discussing them in detail. Depending on the order in which various design activities are performed, we can broadly classify them into two

important stages.

- Preliminary (or high-level) design, and
- Detailed design.

The meaning and scope of these two stages can vary considerably from one design methodology to another. However, for the traditional function-oriented design approach, it is possible to define the objectives of the high-level design as follows:

> Through high-level design, a problem is decomposed into a set of modules. The control relationships among the modules are identified, and also the interfaces among various modules are identified.

The outcome of high-level design is called the program structure or the software architecture. High-level design is a crucial step in the overall design of a software. When the high-level design is complete, the problem should have been decomposed into many small functionally independent modules that are cohesive, have low coupling among themselves, and are arranged in a hierarchy. Many different types of notations have been used to represent a high-level design. A notation that is widely being used for procedural development is a tree-like diagram called the structure chart. Another popular design representation techniques called UML that is being used to document object-oriented design, involves developing several types of diagrams to document the object-oriented design of a systems. Though other notations such as Jackson diagram [1975] or Warnier-Orr [1977, 1981] diagram are available to document a software design, we confine our attention in this text to structure charts and UML diagrams only.

Once the high-level design is complete, detailed design is undertaken.

> During detailed design each module is examined carefully to design its data structures and the algorithms.

The outcome of the detailed design stage is usually documented in the form of a module specification (MSPEC) document. After the high-level design is complete, the problem would have been decomposed into small modules, and the data structures and algorithms to be used described using MSPEC and can be easily grasped by programmers for initiating coding. In this text, we do not discuss MSPECs and confine our attention to high-level design only.

5.1.3 Classification of Design Methodologies

The design activities vary considerably based on the specific design

methodology being used. A large number of software design methodologies are available. We can roughly classify these methodologies into procedural and object-oriented approaches. These two approaches are two fundamentally different design paradigms. In this chapter, we shall discuss the important characteristics of these two fundamental design approaches. Over the next three chapters, we shall study these two approaches in detail.

Do design techniques result in unique solutions?

Even while using the same design methodology, different designers usually arrive at very different design solutions. The reason is that a design technique often requires the designer to make many subjective decisions and work out compromises to contradictory objectives. As a result, it is possible that even the same designer can work out many different solutions to the same problem. Therefore, obtaining a good design would involve trying out several alternatives (or candidate solutions) and picking out the best one. However, a fundamental question that arises at this point is—how to distinguish superior design solution from an inferior one? Unless we know what a good software design is and how to distinguish a superior design solution from an inferior one, we can not possibly design one. We investigate this issue in the next section.

Analysis versus design

Analysis and design activities differ in goal and scope.

> The goal of any analysis technique is to elaborate the customer requirements through careful thinking and at the same time consciously avoiding making any decisions regarding the exact way the system is to be implemented.

The analysis results are generic and does not consider implementation or the issues associated with specific platforms. The analysis model is usually documented using some graphical formalism. In case of the function-oriented approach that we are going to discuss, the analysis model would be documented using data flow diagrams (DFDs), whereas the design would be documented using structure chart. On the other hand, for object-oriented approach, both the design model and the analysis model will be documented using unified modelling language (UML). The analysis model would normally be very difficult to implement using a programming language.

The design model is obtained from the analysis model through

transformations over a series of steps. In contrast to the analysis model, the design model reflects several decisions taken regarding the exact way system is to be implemented. The design model should be detailed enough to be easily implementable using a programming language.

5.2 HOW TO CHARACTERISE A GOOD SOFTWARE DESIGN?

Coming up with an accurate characterisation of a good software design that would hold across diverse problem domains is certainly not easy. In fact, the definition of a "good" software design can vary depending on the exact application being designed. For example, "memory size used up by a program" may be an important issue to Characterise a good solution for embedded software development—since embedded applications are often required to work under severely limited memory sizes due to cost, space, or power consumption considerations. For embedded applications, factors such as design comprehensibility may take a back seat while judging the goodness of design. Thus for embedded applications, one may sacrifice design comprehensibility to achieve code compactness. Similarly, it is not usually true that a criterion that is crucial for some application, needs to be almost completely ignored for another application. It is therefore clear that the criteria used to judge a design solution can vary widely across different types of applications. Not only do the criteria used to judge a design solution depend on the exact application being designed, but to make the matter worse, there is no general agreement among software engineers and researchers on the exact criteria to use for judging a design even for a specific category of application. However, most researchers and software engineers agree on a few desirable characteristics that every good software design for general applications must possess. These characteristics are listed below:

Correctness: A good design should first of all be correct. That is, it should correctly implement all the functionalities of the system.

Understandability: A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.

Efficiency: A good design solution should adequately address resource, time, and cost optimisation issues.

Maintainability: A good design should be easy to change. This is an

important requirement, since change requests usually keep coming from the customer even after product release.

5.2.1 Understandability of a Design: A Ma jor Concern

While performing the design of a certain problem, assume that we have arrived at a large number of design solutions and need to choose the best one. Obviously all incorrect designs have to be discarded first. Out of the correct design solutions, how can we identify the best one?

> Given that we are choosing from only correct design solutions, understandability of a design solution is possibly the most important issue to be considered while judging the goodness of a design.

Recollect from our discussions in Chapter 1 that a good design should help overcome the human cognitive limitations that arise due to limited short-term memory. A large problem overwhelms the human mind, and a poor design would make the matter worse. Unless a design solution is easily understandable, it could lead to an implementation having a large number of defects and at the same time tremendously pushing up the development costs. Therefore, a good design solution should be simple and easily understandable. A design that is easy to understand is also easy to develop and maintain. A complex design would lead to severely increased life cycle costs. Unless a design is easily understandable, it would require tremendous effort to implement, test, debug, and maintain it. We had already pointed out in Chapter 2 that about 60 per cent of the total effort in the life cycle of a typical product is spent on maintenance. If the software is not easy to understand, not only would it lead to increased development costs, the effort required to maintain the product would also increase manifold. Besides, a design solution that is difficult to understand would lead to a program that is full of bugs and is unreliable. Recollect that we had already discussed in Chapter 1 that understandability of a design solution can be enhanced through clever applications of the principles of abstraction and decomposition.

An understandable design is modular and layered

How can the understandability of two different designs be compared, so that we can pick the better one? To be able to compare the understandability of two design solutions, we should at least have an understanding of the general features that an easily understandable design should possess. A design solution should have the following

characteristics to be easily understandable:

- It should assign consistent and meaningful names to various design components.
- It should make use of the principles of decomposition and abstraction in good measures to simplify the design.

We had discussed the essential concepts behind the principles of abstraction and decomposition principles in Chapter 1. But, how can the abstraction and decomposition principles are used in arriving at a design solution? These two principles are exploited by design methodologies to make a design modular and layered. (Though there are also a few other forms in which the abstraction and decomposition principles can be used in the design solution, we discuss those later). We can now define the characteristics of an easily understandable design as follows: A design solution is understandable, if it is modular and the modules are arranged in distinct layers.

A design solution should be modular and layered to be understandable.

We now elaborate the concepts of modularity and layering of modules:

Modularity

A modular design is an effective decomposition of a problem. It is a basic characteristic of any good design solution. A modular design, in simple words, implies that the problem has been decomposed into a set of modules that have only limited interactions with each other. Decomposition of a problem into modules facilitates taking advantage of the divide and conquer principle. If different modules have either no interactions or little interactions with each other, then each module can be understood separately. This reduces the perceived complexity of the design solution greatly. To understand why this is so, remember that it may be very difficult to break a bunch of sticks which have been tied together, but very easy to break the sticks individually.

It is not difficult to argue that modularity is an important characteristic of a good design solution. But, even with this, how can we compare the modularity of two alternate design solutions? From an inspection of the module structure, it is at least possible to intuitively form an idea as to which design is more modular For example, consider two alternate design solutions

to a problem that are represented in Figure 5.2, in which the modules M1 , M2 etc. have been drawn as rectangles. The invocation of a module by another module has been shown as an arrow. It can easily be seen that the design solution of Figure 5.2(a) would be easier to understand since the interactions among the different modules is low. But, can we quantitatively measure the modularity of a design solution? Unless we are able to quantitatively measure the modularity of a design solution, it will be hard to say which design solution is more modular than another. Unfortunately, there are no quantitative metrics available yet to directly measure the modularity of a design. However, we can quantitatively characterise the modularity of a design solution based on the cohesion and coupling existing in the design.

A design solution is said to be highly modular, if the different modules in the solution have high cohesion and their inter-module couplings are low.

A software design with high cohesion and low coupling among modules is the effective problem decomposition we discussed in Chapter 1. Such a design would lead to increased productivity during program development by bringing down the perceived problem complexity.

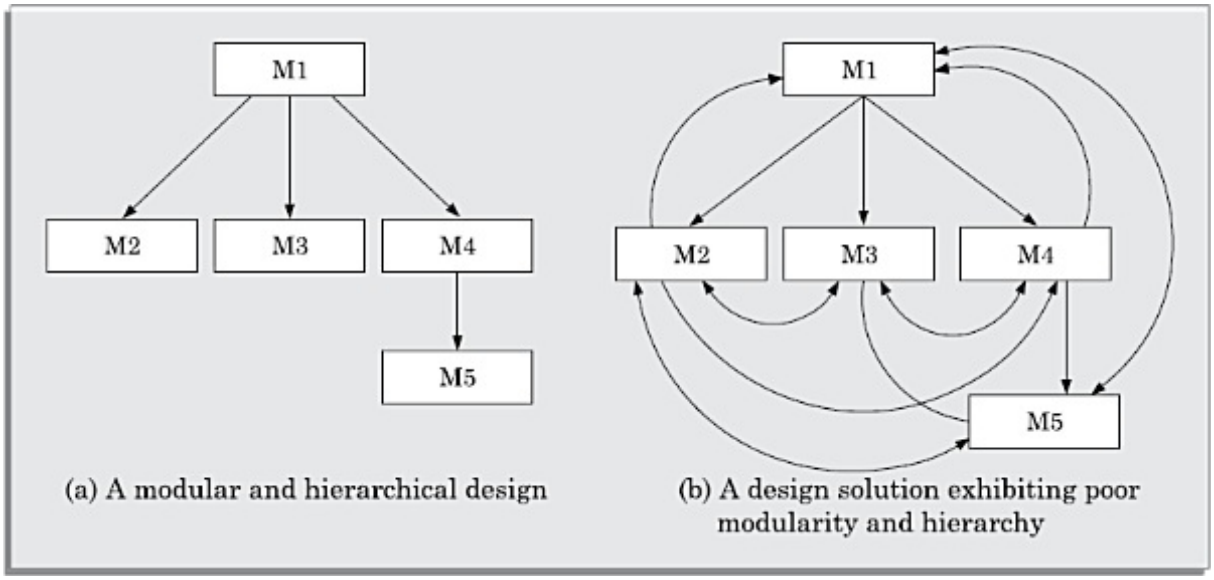

Figure 5.2: Two design solutions to the same problem.

Based on this classification, we would be able to easily judge the cohesion and coupling existing in a design solution. From a knowledge of the cohesion and coupling in a design, we can form our own opinion about the modularity of the design solution. We shall define the concepts of cohesion and coupling and the various classes of cohesion and coupling in Section 5.3. Let us now discuss the other important characteristic of a good design solution—layered

design.

Layered design

A layered design is one in which when the call relations among different modules are represented graphically, it would result in a tree-like diagram with clear layering. In a layered design solution, the modules are arranged in a hierarchy of layers. A module can only invoke functions of the modules in the layer immediately below it. The higher layer modules can be considered to be similar to managers that invoke (order) the lower layer modules to get certain tasks done. A layered design can be considered to be implementing control abstraction, since a module at a lower layer is unaware of (about how to call) the higher layer modules.

A layered design can make the design solution easily understandable, since to understand the working of a module, one would at best have to understand how the immediately lower layer modules work without having to worry about the functioning of the upper layer modules.

When a failure is detected while executing a module, it is obvious that the modules below it can possibly be the source of the error. This greatly simplifies debugging since one would need to concentrate only on a few modules to detect the error. We shall elaborate these concepts governing layered design of modules in Section 5.4.

5.3 COHESION AND COUPLING

We have so far discussed that effective problem decomposition is an important characteristic of a good design. Good module decomposition is indicated through high cohesion of the individual modules and low coupling of the modules with each other. Let us now define what is meant by cohesion and coupling.

Cohesion is a measure of the functional strength of a module, whereas the coupling between two modules is a measure of the degree of interaction (or interdependence) between the two modules.

In this section, we first elaborate the concepts of cohesion and coupling. Subsequently, we discuss the classification of cohesion and coupling.

Coupling: Intuitively, we can think of coupling as follows. Two modules are said to be highly coupled, if either of the following two situations arise:

- If the function calls between two modules involve passing large chunks

of shared data, the modules are tightly coupled.

- If the interactions occur through some shared data, then also we say that they are highly coupled.

If two modules either do not interact with each other at all or at best interact by passing no data or only a few primitive data items, they are said to have low coupling.

Cohesion: To understand cohesion, let us first understand an analogy. Suppose you listened to a talk by some speaker. You would call the speech to be cohesive, if all the sentences of the speech played some role in giving the talk a single and focused theme. Now, we can extend this to a module in a design solution. When the functions of the module co-operate with each other for performing a single objective, then the module has good cohesion. If the functions of the module do very different things and do not co-operate with each other to perform a single piece of work, then the module has very poor cohesion.

Functional independence

By the term functional independence, we mean that a module performs a single task and needs very little interaction with other modules.

> A module that is highly cohesive and also has low coupling with other modules is said to be functionally independent of the other modules.

Functional independence is a key to any good design primarily due to the following advantages it offers:

Error isolation: Whenever an error exists in a module, functional independence reduces the chances of the error propagating to the other modules. The reason behind this is that if a module is functionally independent, its interaction with other modules is low. Therefore, an error existing in the module is very unlikely to affect the functioning of other modules.

Further, once a failure is detected, error isolation makes it very easy to locate the error. On the other hand, when a module is not functionally independent, once a failure is detected in a functionality provided by the module, the error can be potentially in any of the large number of modules and propagated to the functioning of the module.

Scope of reuse: Reuse of a module for the development of other applications becomes easier. The reasons for this is as follows. A functionally

independent module performs some well-defined and precise task and the interfaces of the module with other modules are very few and simple. A functionally independent module can therefore be easily taken out and reused in a different program. On the other hand, if a module interacts with several other modules or the functions of a module perform very different tasks, then it would be difficult to reuse it. This is especially so, if the module accesses the data (or code) internal to other modules.

Understandability: When modules are functionally independent, complexity of the design is greatly reduced. This is because of the fact that different modules can be understood in isolation, since the modules are independent of each other. We have already pointed out in Section 5.2 that understandability is a major advantage of a modular design. Besides the three we have listed here, there are many other advantages of a modular design as well. We shall not list those here, and leave it as an assignment to the reader to identify them.

5.3.1 Classification of Cohesiveness

Cohesiveness of a module is the degree to which the different functions of the module co-operate to work towards a single objective. The different modules of a design can possess different degrees of freedom. However, the different classes of cohesion that modules can possess are depicted in Figure 5.3. The cohesiveness increases from coincidental to functional cohesion. That is, coincidental is the worst type of cohesion and functional is the best cohesion possible. These different classes of cohesion are elaborated below.

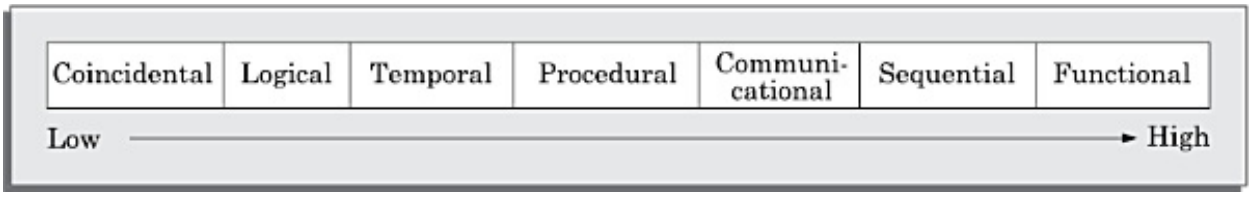

Figure 5.3: Classification of cohesion.

Coincidental cohesion: A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, we can say that the module contains a random collection of functions. It is likely that the functions have been placed in the module out of pure coincidence rather than through some thought or design. The designs made by novice programmers often possess this category of cohesion, since they often bundle functions to modules rather arbitrarily. An example of a module with coincidental cohesion

has been shown in Figure 5.4(a).Observe that the different functions of the module carry out very different and unrelated activities starting from issuing of library books to creating library member records on one hand, and handling librarian leave request on the other.

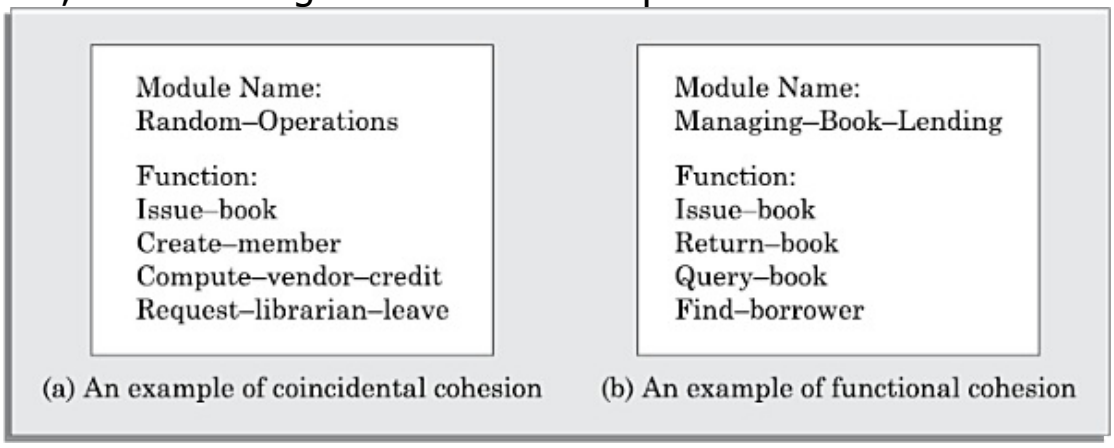

Figure 5.4: Examples of cohesion.

Logical cohesion: A module is said to be logically cohesive, if all elements of the module perform similar operations, such as error handling, data input, data output, etc. As an example of logical cohesion, consider a module that contains a set of print functions to generate various types of output reports such as grade sheets, salary slips, annual reports, etc.

Temporal cohesion: When a module contains functions that are related by the fact that these functions are executed in the same time span, then the module is said to possess temporal cohesion. As an example, consider the following situation. When a computer is booted, several functions need to be performed. These include initialisation of memory and devices, loading the operating system, etc. When a single module performs all these tasks, then the module can be said to exhibit temporal cohesion. Other examples of modules having temporal cohesion are the following. Similarly, a module would exhibit temporal cohesion, if it comprises functions for performing initialisation, or start-up, or shut-down of some process.

Procedural cohesion: A module is said to possess procedural cohesion, if the set of functions of the module are executed one after the other, though these functions may work towards entirely different purposes and operate on very different data. Consider the activities associated with order processing in a trading house. The functions login(), place-order(), check-order(), print-bill(), place-order-on-vendor(), update-inventory(), and logout() all do different thing and operate on different data. However, they are normally

executed one after the other during typical order processing by a sales clerk.

Communicational cohesion: A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure. As an example of procedural cohesion, consider a module named student in which the different functions in the module such as admitStudent, enterMarks, printGradeSheet, etc. access and manipulate data stored in an array named studentRecords defined within the module.

Sequential cohesion: A module is said to possess sequential cohesion, if the different functions of the module execute in a sequence, and the output from one function is input to the next in the sequence. As an example consider the following situation. In an on-line store consider that after a customer requests for some item, it is first determined if the item is in stock. In this case, if the functions create-order(), check-item-availability(), place-order-on-vendor() are placed in a single module, then the module would exhibit sequential cohesion. Observe that the function create-order() creates an order that is processed by the function check-item-availability() (whether the items are available in the required quantities in the inventory) is input to place-order-on-vendor().

Functional cohesion: A module is said to possess functional cohesion, if different functions of the module co-operate to complete a single task. For example, a module containing all the functions required to manage employees' pay-roll displays functional cohesion. In this case, all the functions of the module (e.g., computeOvertime(), computeWorkHours(), computeDeductions(), etc.) work together to generate the payslips of the employees. Another example of a module possessing functional cohesion has been shown in Figure 5.4(b). In this example, the functions issue-book(), return-book(), query-book(), and find-borrower(), together manage all activities concerned with book lending. When a module possesses functional cohesion, then we should be able to describe what the module does using only one simple sentence. For example, for the module of Figure 5.4(a), we can describe the overall responsibility of the module by saying "It manages the book lending procedure of the library."

A simple way to determine the cohesiveness of any given module is as follows. First examine what do the functions of the module perform. Then, try to write down a sentence to describe the overall work performed by the module. If you need a compound sentence to describe the functionality of the module, then it has sequential or communicational cohesion. If you need words such as "first", "next", "after", "then", etc., then it possesses sequential

or temporal cohesion. If it needs words such as "initialise", "setup", "shut down", etc., to define its functionality, then it has temporal cohesion.

We can now make the following observation. A cohesive module is one in which the functions interact among themselves heavily to achieve a single goal. As a result, if any of these functions is removed to a different module, the coupling would increase as the functions would now interact across two different modules.

5.3.2 Classification of Coupling

The coupling between two modules indicates the degree of interdependence between them. Intuitively, if two modules interchange large amounts of data, then they are highly interdependent or coupled. We can alternately state this concept as follows.

The degree of coupling between two modules depends on their interface complexity.

The interface complexity is determined based on the number of parameters and the complexity of the parameters that are interchanged while one module invokes the functions of the other module.

Let us now classify the different types of coupling that can exist between two modules. Between any two interacting modules, any of the following five different types of coupling can exist. These different types of coupling, in increasing order of their severities have also been shown in Figure 5.5.

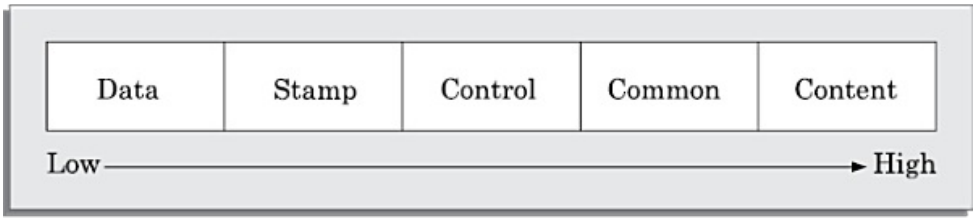

Figure 5.5: Classification of coupling.

Data coupling: Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between the two, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for control purposes.

Stamp coupling: Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.

Control coupling: Control coupling exists between two modules, if data from one module is used to direct the order of instruction execution in another. An example of control coupling is a flag set in one module and

tested in another module.

Common coupling: Two modules are common coupled, if they share some global data items.

Content coupling: Content coupling exists between two modules, if they share code. That is, a jump from one module into the code of another module can occur. Modern high-level programming languages such as C do not support such jumps across modules.

The different types of coupling are shown schematically in Figure 5.5. The degree of coupling increases from data coupling to content coupling. High coupling among modules not only makes a design solution difficult to understand and maintain, but it also increases development effort and also makes it very difficult to get these modules developed independently by different team members.

5.4 LAYERED ARRANGEMENT OF MODULES

The control hierarchy represents the organisation of program components in terms of their call relationships. Thus we can say that the control hierarchy of a design is determined by the order in which different modules call each other. Many different types of notations have been used to represent the control hierarchy. The most common notation is a tree-like diagram known as a structure chart which we shall study in some detail in Chapter 6. However, other notations such as Warnier-Orr [1977, 1981] or Jackson diagrams [1975] may also be used. Since, Warnier-Orr and Jackson's notations are not widely used nowadays, we shall discuss only structure charts in this text.

In a layered design solution, the modules are arranged into several layers based on their call relationships. A module is allowed to call only the modules that are at a lower layer. That is, a module should not call a module that is either at a higher layer or even in the same layer. Figure 5.6(a) shows a layered design, whereas Figure 5.6(b) shows a design that is not layered. Observe that the design solution shown in Figure 5.6(b), is actually not layered since all the modules can be considered to be in the same layer. In the following, we state the significance of a layered design and subsequently we explain it.

An important characteristic feature of a good design solution is layering of the modules. A layered design achieves control abstraction and is easier to understand and debug.

In a layered design, the top-most module in the hierarchy can be considered as a manager that only invokes the services of the lower level module to discharge its responsibility. The modules at the intermediate layers offer services to their higher layer by invoking the services of the lower layer modules and also by doing some work themselves to a limited extent. The modules at the lowest layer are the worker modules. These do not invoke services of any module and entirely carry out their responsibilities by themselves.

Understanding a layered design is easier since to understand one module, one would have to at best consider the modules at the lower layers (that is, the modules whose services it invokes). Besides, in a layered design errors are isolated, since an error in one module can affect only the higher layer modules. As a result, in case of any failure of a module, only the modules at the lower levels need to be investigated for the possible error. Thus, debugging time reduces significantly in a layered design. On the other hand, if the different modules call each other arbitrarily, then this situation would correspond to modules arranged in a single layer. Locating an error would be both difficult and time consuming. This is because, once a failure is observed, the cause of failure (i.e. error) can potentially be in any module, and all modules would have to be investigated for the error. In the following, we discuss some important concepts and terminologies associated with a layered design:

Superordinate and subordinate modules: In a control hierarchy, a module that controls another module is said to be superordinate to it. Conversely, a module controlled by another module is said to be subordinate to the controller.

Visibility: A module B is said to be visible to another module A, if A directly calls B. Thus, only the immediately lower layer modules are said to be visible to a module.

Control abstraction: In a layered design, a module should only invoke the functions of the modules that are in the layer immediately below it. In other words, the modules at the higher layers, should not be visible (that is, abstracted out) to the modules at the lower layers. This is referred to as control abstraction.

Depth and width: Depth and width of a control hierarchy provide an indication of the number of layers and the overall span of control respectively. For the design of Figure 5.6(a), the depth is 3 and width is also 3.

Fan-out: Fan-out is a measure of the number of modules that are directly controlled by a given module. In Figure 5.6(a), the fan-out of the module M1 is 3. A design in which the modules have very high fan-out numbers is not a good design. The reason for this is that a very high fan-out is an indication that the module lacks cohesion. A module having a large fan-out (greater than 7) is likely to implement several different functions and not just a single cohesive function.

Fan-in: Fan-in indicates the number of modules that directly invoke a given module. High fan-in represents code reuse and is in general, desirable in a good design. In Figure 5.6(a), the fan-in of the module M1 is 0, that of M2 is 1, and that of M5 is 2.

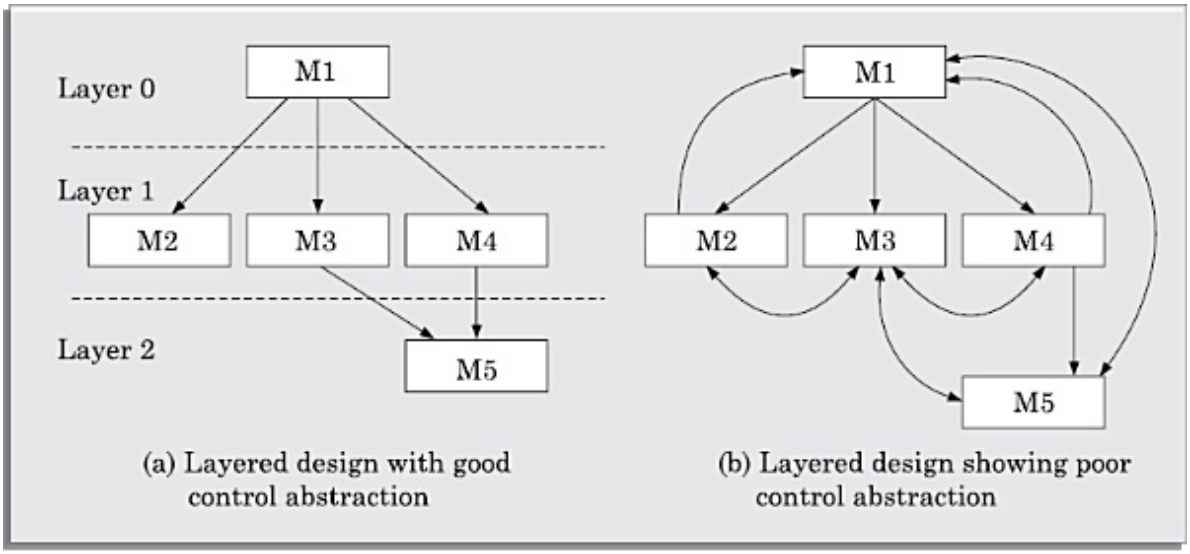

Figure 5.6: Examples of good and poor control abstraction.

5.5 APPROACHES TO SOFTWARE DESIGN

There are two fundamentally different approaches to software design that are in use today— function-oriented design, and object-oriented design. Though these two design approaches are radically different, they are complementary rather than competing techniques. The object-oriented approach is a relatively newer technology and is still evolving. For development of large programs, the object- oriented approach is becoming increasingly popular due to certain advantages that it offers. On the other hand, function-oriented designing is a mature technology and has a large following. Salient features of these two approaches are discussed in subsections 5.5.1 and 5.5.2 respectively.

5.5.1 Function-oriented Design

The following are the salient features of the function-oriented design approach:

Top-down decomposition: A system, to start with, is viewed as a black box that provides certain services (also known as high-level functions) to the users of the system.

In top-down decomposition, starting at a high-level view of the system, each high-level function is successively refined into more detailed functions.

For example, consider a function create-new-library member which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge. This high-level function may be refined into the following subfunctions:

- assign-membership-number
- create-member-record
- print-bill

Each of these subfunctions may be split into more detailed subfunctions and so on.

Centralised system state: The system state can be defined as the values of certain data items that determine the response of the system to a user action or external event. For example, the set of books (i.e. whether borrowed by different users or available for issue) determines the state of a library automation system. Such data in procedural programs usually have global scope and are shared by many modules.

The system state is centralised and shared among different functions.

For example, in the library management system, several functions such as the following share data such as member-records for reference and updation:

- create-new-member
- delete-member
- update-member-record

A large number of function-oriented design approaches have been proposed in the past. A

few of the well-established function-oriented design approaches are as following:

- Structured design by Constantine and Yourdon, [1979]
- Jackson's structured design by Jackson [1975]
- Warnier-Orr methodology [1977, 1981]

- Step-wise refinement by Wirth [1971]
- Hatley and Pirbhai's Methodology [1987]

5.5.2 Object-oriented Design

In the object-oriented design (OOD) approach, a system is viewed as being made up of a collection of objects (i.e. entities). Each object is associated with a set of functions that are called its methods. Each object contains its own data and is responsible for managing it. The data internal to an object cannot be accessed directly by other objects and only through invocation of the methods of the object. The system state is decentralised since there is no globally shared data in the system and data is stored in each object. For example, in a library automation software, each library member may be a separate object with its own data and functions to operate on the stored data. The methods defined for one object cannot directly refer to or change the data of other objects.

The object-oriented design paradigm makes extensive use of the principles of abstraction and decomposition as explained below. Objects decompose a system into functionally independent modules. Objects can also be considered as instances of abstract data types (ADTs). The ADT concept did not originate from the object-oriented approach. In fact, ADT concept was extensively used in the ADA programming language introduced in the 1970s. ADT is an important concept that forms an important pillar of object-orientation. Let us now discuss the important concepts behind an ADT. There are, in fact, three important concepts associated with an ADT—data abstraction, data structure, data type. We discuss these in the following subsection:

Data abstraction: The principle of data abstraction implies that how data is exactly stored is abstracted away. This means that any entity external to the object (that is, an instance of an ADT) would have no knowledge about how data is exactly stored, organised, and manipulated inside the object. The entities external to the object can access the data internal to an object only by calling certain well-defined methods supported by the object. Consider an ADT such as a stack. The data of a stack object may internally be stored in an array, a linearly linked list, or a bidirectional linked list. The external entities have no knowledge of this and can access data of a stack object only through the supported operations such as push and pop.

Data structure: A data structure is constructed from a collection of primitive data items. Just as a civil engineer builds a large civil engineering structure using primitive building materials such as bricks, iron rods, and cement; a programmer can construct a data structure as an organised collection of primitive data items such as integer, floating point numbers, characters, etc.

Data type: A type is a programming language terminology that refers to anything that can be instantiated. For example, int, float, char etc., are the basic data types supported by C programming language. Thus, we can say that ADTs are user defined data types.

In object-orientation, classes are ADTs. But, what is the advantage of developing an application using ADTs? Let us examine the three main advantages of using ADTs in programs:

- The data of objects are encapsulated within the methods. The encapsulation principle is also known as data hiding. The encapsulation principle requires that data can be accessed and manipulated only through the methods supported by the object and not directly. This localises the errors. The reason for this is as follows. No program element is allowed to change a data, except through invocation of one of the methods. So, any error can easily be traced to the code segment changing the value. That is, the method that changes a data item, making it erroneous can be easily identified.
- An ADT-based design displays high cohesion and low coupling. Therefore, object- oriented designs are highly modular.
- Since the principle of abstraction is used, it makes the design solution easily understandable and helps to manage complexity.

Similar objects constitute a class. In other words, each object is a member of some class. Classes may inherit features from a super class. Conceptually, objects communicate by message passing. Objects have their own internal data. Thus an object may exist in different states depending the values of the internal data. In different states, an object may behave differently. We shall elaborate these concepts in Chapter 7 and subsequently we discuss an object-oriented design methodology in Chapter 8.

Object-oriented versus function-oriented design approaches

The following are some of the important differences between the

function-oriented and object-oriented design:

- Unlike function-oriented design methods in OOD, the basic abstraction is not the services available to the users of the system such as issue-book, display-book-details, find-issued-books, etc., but real-world entities such as member, book, book-register, etc. For example in OOD, an employee pay-roll software is not developed by designing functions such as update-employee-record, get-employee-address, etc., but by designing objects such as employees, departments, etc.
- In OOD, state information exists in the form of data distributed among several objects of the system. In contrast, in a procedural design, the state information is available in a centralised shared data store. For example, while developing an employee pay-roll system, the employee data such as the names of the employees, their code numbers, basic salaries, etc., are usually implemented as global data in a traditional programming system; whereas in an object-oriented design, these data are distributed among different employee objects of the system. Objects communicate by message passing. Therefore, one object may discover the state information of another object by sending a message to it. Of course, somewhere or other the real-world functions must be implemented.
- Function-oriented techniques group functions together if, as a group, they constitute a higher level function. On the other hand, object-oriented techniques group functions together on the basis of the data they operate on.

To illustrate the differences between the object-oriented and the function-oriented design approaches, let us consider an example—that of an automated fire-alarm system for a large building.

Automated fire-alarm system—customer requirements

The owner of a large multi-storied building wants to have a computerised fire alarm system designed, developed, and installed in his building. Smoke detectors and fire alarms would be placed in each room of the building. The fire alarm system would monitor the status of these smoke detectors. Whenever a fire condition is reported by any of the smoke detectors, the fire alarm system should determine the location at which the fire has been sensed and then sound the alarms

only in the neighbouring locations. The fire alarm system should also flash an alarm message on the computer console. Fire fighting personnel would man the console round the clock. After a fire condition has been successfully handled, the fire alarm system should support resetting the alarms by the fire fighting personnel.

Function-oriented approach: In this approach, the different high-level functions are first identified, and then the data structures are designed.

```
/* Global data (system state) accessible by various functions */
        BOOL  detector_status[MAX_ROOMS];
        int   detector_locs[MAX_ROOMS];
        BOOL  alarm-status[MAX_ROOMS]; /* alarm activated when status is set */
        int   alarm_locs[MAX_ROOMS]; /* room number where alarm is located */
        int   neighbour-alarms[MAX-ROOMS][10]; /* each detector has at most */
                                               /* 10 neighbouring alarm locations */
        int   sprinkler[MAX_ROOMS];
```

The functions which operate on the system state are:

```
interrogate_detectors();
get_detector_location();
determine_neighbour_alarm();
determine_neighbour_sprinkler();
ring_alarm();
activate_sprinkler();
reset_alarm();
reset_sprinkler();
report_fire_location();
```

Object-oriented approach: In the object-oriented approach, the different classes of objects are identified. Subsequently, the methods and data for each object are identified. Finally, an appropriate number of instances of each class is created.

```
class detector
attributes: status, location, neighbours
operations: create, sense-status, get-location,
       find-neighbours

class alarm
attributes: location, status
operations: create, ring-alarm, get_location, reset-
alarm

class sprinkler
```

```
attributes: location, status
operations: create, activate-sprinkler, get_location,
reset-sprinkler
```

We can now compare the function-oriented and the object-oriented approaches based on the two examples discussed above, and easily observe the following main differences:

- In a function-oriented program, the system state (data) is centralised and several functions access and modify this central data. In case of an object-oriented program, the state information (data) is distributed among various objects.
- In the object-oriented design, data is private in different objects and these are not available to the other objects for direct access and modification.
- The basic unit of designing an object-oriented program is objects, whereas it is functions and modules in procedural designing. Objects appear as nouns in the problem description; whereas functions appear as verbs.

At this point, we must emphasise that it is not necessary that an object-oriented design be implemented by using an object-oriented language only. However, an object-oriented language such as C++ and Java support the definition of all the basic mechanisms of class, inheritance, objects, methods, etc. and also support all key object-oriented concepts that we have just discussed. Thus, an object-oriented language facilitates the implementation of an OOD. However, an OOD can as well be implemented using a conventional procedural languages—though it may require more effort to implement an OOD using a procedural language as compared to the effort required for implementing the same design using an object-oriented language. In fact, the older C++ compilers were essentially pre-processors that translated C++ code into C code.

Even though object-oriented and function-oriented techniques are remarkably different approaches to software design, yet one does not replace the other; but they complement each other in some sense. For example, usually one applies the top-down function oriented techniques to design the internal methods of a class, once the classes are identified. In this case, though outwardly the system appears to have been developed in an object-oriented fashion, but inside each class there may be a small hierarchy of

functions designed in a top-down manner.

SUMMARY

- software design is typically carried out through two stages—high-level design, and detailed design. During high-level design, the important components (modules) of the system and their interactions are identified. During detailed design, the algorithms and data structures are identified.
- We discussed that there is no unique design solution to any problem and one needs to choose the best solution among a set of candidate solutions. To be able to achieve this, we identified the factors based on which a superior design can be distinguished from a inferior design.
- We discussed that understandability of a design is a major criterion determining the goodness of a design. We Characterised the understandability of design in terms of satisfactory usage of decomposition and abstraction principles. Later, we Characterised these in terms of cohesion, coupling, layering, control abstraction, fan-in, fan-out, etc.
- We identified two fundamentally different approaches to software design—function- oriented design and object-oriented design. We discussed the essential philosophy governing these two approaches and argued that these two approaches to software design are not really competing approaches but complementary approaches.

EXERCISES

1. Choose the correct option
   (a) The extent of data interchange between two modules is called:
      (i) Coupling
      (ii) Cohesion
      (iii) Structure
      (iv) Union
   (b) Which of the following type of cohesion can be considered as the strongest cohesion:
      (i) Logical
      (ii) Coincidental
      (iii) Temporal
      (iv) Functional

(c) The modules in a good software design should have which of the following characteristics:
(i) High cohesion, low coupling
(ii) Low cohesion, high coupling
(iii) Low cohesion, low coupling
(iv) High cohesion, high coupling

2. Do you agree with the following assertion? A design solution that is difficult to under- stand would lead to increased development and maintenance cost. Give reasonings for your answer.
3. What do you mean by the terms cohesion and coupling in the context of software design?
How are these concepts useful in arriving at a good design of a system?
4. What do you mean by a modular design? How can you determine whether a given design is modular or not?
5. Enumerate the different types of cohesion that a module in a design might exhibit. Give examples of each.
6. Enumerate the different types of coupling that might exist between two modules. Give examples of each.
7. Is it true that whenever you increase the cohesion of your design, coupling in the design would automatically decrease? Justify your answer by using suitable examples.
8. What according to you are the characteristics of a good software design?
9. What do you understand by the term functional independence in the context of software design? What are the advantages of functional independence? How can functional independence in a software design be achieved?
10. Explain how the principles of abstraction and decomposition are used to arrive at a good design.
11. What do you understand by information hiding in the context of software design?
Explain why a design approach based on the information hiding principle is likely to lead to a reusable and maintainable design. Illustrate your answer with a suitable example.
12. In the context of software development, distinguish between analysis and design with respect to intention, methodology, and the documentation technique used.
13. State whether the following statements are **TRUE** o r **FALSE**. Give reasons for your answer.

(a) The essence of any good function-oriented design technique is to map the functions performing similar activities into a module.
(b) Traditional procedural design is carried out top-down whereas object-oriented design is normally carried out bottom-up.
(c) Common coupling is the worst type of coupling between two modules.
(d) Temporal cohesion is the worst type of cohesion that a module can have.
(e) The extent to which two modules depend on each other determines the cohesion of the two modules.

14. Compare relative advantages of the object-oriented and function-oriented approaches to software design.
15. Name a few well-established function-oriented software design techniques.
16. Explain the important causes of and remedies for high coupling between two software modules.
17. What problems are likely to arise if two modules have high coupling?
18. What problems are likely to occur if a module has low cohesion?
19. Distinguish between high-level and detailed designs. What documents should be produced on completion of high-level and detailed designs respectively?
20. What is meant by the term cohesion in the context of software design? Is it true that in a good design, the modules should have low cohesion? Why?
21. What is meant by the term coupling in the context of software design? Is it true that in a good design, the modules should have low coupling? Why?
22. What do you mean by modular design? What are the different factors that affect the modularity of a design? How can you assess the modularity of a design? What are the advantages of a modular design?
23. How would you improve a software design that displays very low cohesion and high coupling?
24. Explain how the overall cohesion and coupling of a design would be impacted if all modules of the design are merged into a single module.
25. Explain what do you understand by the terms decomposition and abstraction in the context of software design. How are these two principles used in arriving good procedural designs?
26. What is an ADT? What advantages accrue when a software design

technique is based on ADTs? Explain why the object paradigm is said to be based on ADTs.

27. By using suitable examples explain the following terms associated with an abstract data type (ADT)—data abstraction, data structure, data type.
28. What do you understand by the term top-down decomposition in the context of function- oriented design? Explain your answer using a suitable example.
29. What do you understand by a layered software design? What are the advantages of a layered design? Explain your answer by using suitable examples.
30. What is the principal difference between the software design methodologies based on functional abstraction and those based on data abstraction? Name at least one popular design technique based on each of these two software design paradigms.
31. What are the main advantages of using an object-oriented approach to software design over a function-oriented approach?
32. Point out three important differences between the function oriented and the object- oriented approaches to software design. Corroborate your answer through suitable examples.
33. Identify the criteria that you would use to decide which one of two alternate function- oriented design solutions to a problem is superior.
34. Explain the main differences between architectural design, high-level-design, and detailed design of a software system.

Chapter 6

FUNCTION-ORIENTED SOFTWARE DESIGN

Function-oriented design techniques were proposed nearly four decades ago. These techniques are at the present time still very popular and are currently being used in many software development organisations. These techniques, to start with, view a system as a black-box that provides a set of services to the users of the software. These services provided by a software (e.g., issue book, serach book, etc., for a Library Automation Software to its users are also known as the high-level functions supported by the software. During the design process, these high-level functions are successively decomposed into more detailed functions.

> The term top-down decomposition is often used to denote the successive decomposition of a set of high-level functions into more detailed functions.

After top-down decomposition has been carried out, the different identified functions are mapped to modules and a module structure is created. This module structure would possess all the characteristics of a good design identified in the last chapter.

In this text, we shall not focus on any specific design methodology. Instead, we shall discuss a methodology that has the essential features of several important function-oriented design methodologies. Such an approach shall enable us to easily assimilate any specific design methodology in the future whenever the need arises. Learning a specific methodology may become necessary for you later, since different software development houses follow different methodologies. After all, the different procedural design techniques can be considered as sister techniques that have only minor differences with respect to the methodology and notations. We shall call the design technique discussed in this text as structured analysis/structured design (SA/SD)

methodology. This technique draws heavily from the design methodologies proposed by the following authors:

- DeMarco and Yourdon [1978]
- Constantine and Yourdon [1979]
- Gane and Sarson [1979]
- Hatley and Pirbhai [1987]

The SA/SD technique can be used to perform the high-level design of a software. The details of SA/SD technique are discussed further.

6.1 OVERVIEW OF SA/SD METHODOLOGY

As the name itself implies, SA/SD methodology involves carrying out two distinct activities:

- Structured analysis (SA)
- Structured design (SD)

The roles of structured analysis (SA) and structured design (SD) have been shown schematically in Figure 6.1. Observe the following from the figure:

- During structured analysis, the SRS document is transformed into a data flow diagram (DFD) model.
- During structured design, the DFD model is transformed into a structure chart.

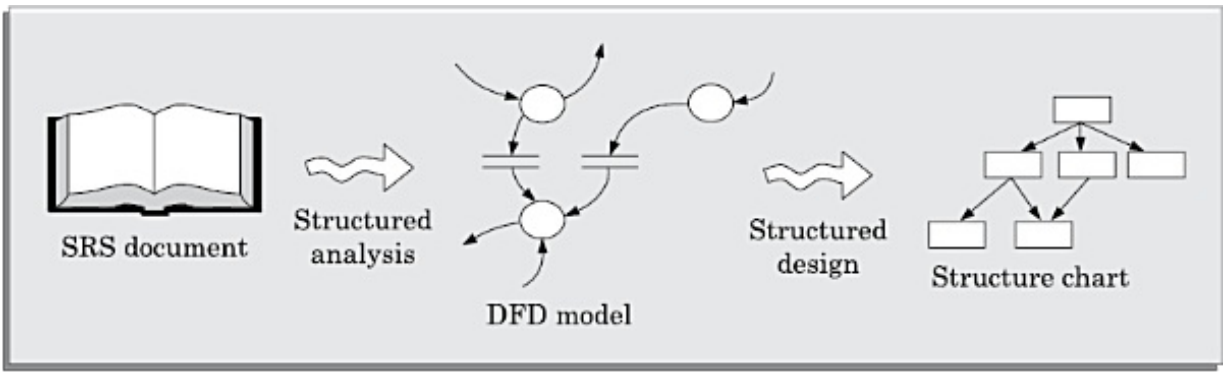

Figure 6.1: Structured analysis and structured design methodology.

As shown in Figure 6.1, the structured analysis activity transforms the SRS document into a graphic model called the DFD model. During structured analysis, functional decomposition of the system is achieved. That is, each

function that the system needs to perform is analysed and hierarchically decomposed into more detailed functions. On the other hand, during structured design, all functions identified during structured analysis are mapped to a module structure. This module structure is also called the high-level design or the software architecture for the given problem. This is represented using a structure chart.

The high-level design stage is normally followed by a detailed design stage. During the detailed design stage, the algorithms and data structures for the individual modules are designed. The detailed design can directly be implemented as a working system using a conventional programming language.

> It is important to understand that the purpose of structured analysis is to capture the detailed structure of the system as perceived by the user, whereas the purpose of structured design is to define the structure of the solution that is suitable for implementation in some programming language.

The results of structured analysis can therefore, be easily understood by the user. In fact, the different functions and data in structured analysis are named using the user's terminology. The user can therefore even review the results of the structured analysis to ensure that it captures all his requirements.

In the following section, we first discuss how to carry out structured analysis to construct the DFD model. Subsequently, we discuss how the DFD model can be transformed into structured design.

6.2 STRUCTURED ANALYSIS

We have already mentioned that during structured analysis, the major processing tasks (high-level functions) of the system are analysed, and the data flow among these processing tasks are represented graphically. Significant contributions to the development of the structured analysis techniques have been made by Gane and Sarson [1979], and DeMarco and Yourdon [1978]. The structured analysis technique is based on the following underlying principles:

- Top-down decomposition approach.
- Application of divide and conquer principle. Through this each high-level function is independently decomposed into detailed functions.
- Graphical representation of the analysis results using data flow

diagrams (DFDs).

DFD representation of a problem, as we shall see shortly, is very easy to construct. Though extremely simple, it is a very powerful tool to tackle the complexity of industry standard problems.

> A DFD is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among those functions.

Please note that a DFD model only represents the data flow aspects and does not show the sequence of execution of the different functions and the conditions based on which a function may or may not be executed. In fact, it completely ignores aspects such as control flow, the specific algorithms used by the functions, etc. In the DFD terminology, each function is called a process or a bubble. It is useful to consider each function as a processing station (or process) that consumes some input data and produces some output data.

DFD is an elegant modelling technique that can be used not only to represent the results of structured analysis of a software problem, but also useful for several other applications such as showing the flow of documents or items in an organisation. Recall that in Chapter 1 we had given an example (see Figure 1.10) to illustrate how a DFD can be used to represent the processing activities and flow of material in an automated car assembling plant. We now elaborate how a DFD model can be constructed.

6.2.1 Data Flow Diagrams (DFDs)

The DFD (also known as the bubble chart) is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on those data, and the output data generated by the system. The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism— it is simple to understand and use. A DFD model uses a very limited number of primitive symbols (shown in Figure 6.2) to represent the functions performed by a system and the data flow among these functions.

Starting with a set of high-level functions that a system performs, a DFD model represents the subfunctions performed by the functions using a hierarchy of diagrams. We had pointed out while discussing the principle of abstraction in Section 1.3.2 that any hierarchical representation is an

effective means to tackle complexity. Human mind is such that it can easily understand any hierarchical model of a system—because in a hierarchical model, starting with a very abstract model of a system, various details of the system are slowly introduced through different levels of the hierarchy. The DFD technique is also based on a very simple set of intuitive concepts and rules. We now elaborate the different concepts associated with building a DFD model of a system.

Primitive symbols used for constructing DFDs

There are essentially five different types of symbols used for constructing DFDs. These primitive symbols are depicted in Figure 6.2. The meaning of these symbols are explained as follows:

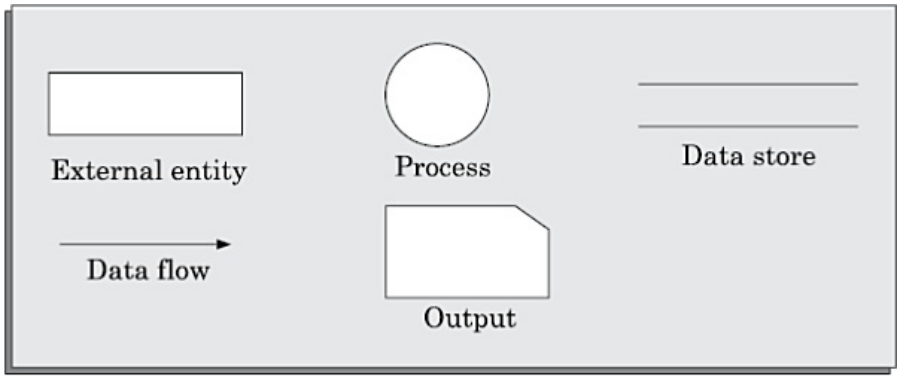

Figure 6.2: Symbols used for designing DFDs.

Function symbol: A function is represented using a circle. This symbol is called a process or a bubble. Bubbles are annotated with the names of the corresponding functions (see Figure 6.3).

External entity symbol: An external entity such as a librarian, a library member, etc. is represented by a rectangle. The external entities are essentially those physical entities external to the software system which interact with the system by inputting data to the system or by consuming the data produced by the system. In addition to the human users, the external entity symbols can be used to represent external hardware and software such as another application software that would interact with the software being modelled.

Data flow symbol: A directed arc (or an arrow) is used as a data flow symbol. A data flow symbol represents the data flow occurring between two processes or between an external entity and a process in the direction of the data flow arrow. Data flow symbols are usually annotated with the corresponding data names. For example the DFD in Figure 6.3(a) shows three data flows—the

data item number flowing from the process read-number to validate-number, data-item flowing into read-number, and valid-number flowing out of validate-number.

Data store symbol: A data store is represented using two parallel lines. It represents a logical file. That is, a data store symbol can represent either a data structure or a physical file on disk. Each data store is connected to a process by means of a data flow symbol. The direction of the data flow arrow shows whether data is being read from or written into a data store. An arrow flowing in or out of a data store implicitly represents the entire data of the data store and hence arrows connecting to a data store need not be annotated with the name of the corresponding data items. As an example of a data store, number is a data store in Figure 6.3(b).

Output symbol: The output symbol is as shown in Figure 6.2. The output symbol is used when a hard copy is produced.

The notations that we are following in this text are closer to the Yourdon's notations than to the other notations. You may sometimes find notations in other books that are slightly different than those discussed here. For example, the data store may look like a box with one end open. That is because, they may be following notations such as those of Gane and Sarson [1979].

Important concepts associated with constructing DFD models

Before we discuss how to construct the DFD model of a system, let us discuss some important concepts associated with DFDs:

Synchronous and asynchronous operations

If two bubbles are directly connected by a data flow arrow, then they are synchronous. This means that they operate at the same speed. An example of such an arrangement is shown in Figure 6.3(a). Here, the `validate-number` bubble can start processing only after the read-number bubble has supplied data to it; and the `read-number` bubble has to wait until the `validate-number` bubble has consumed its data.

However, if two bubbles are connected through a data store, as in Figure 6.3(b) then the speed of operation of the bubbles are independent. This statement can be explained using the following reasoning. The data produced by a producer bubble gets stored in the data store. It is therefore possible that the producer bubble stores several pieces of data items, even before the

consumer bubble consumes any of them.

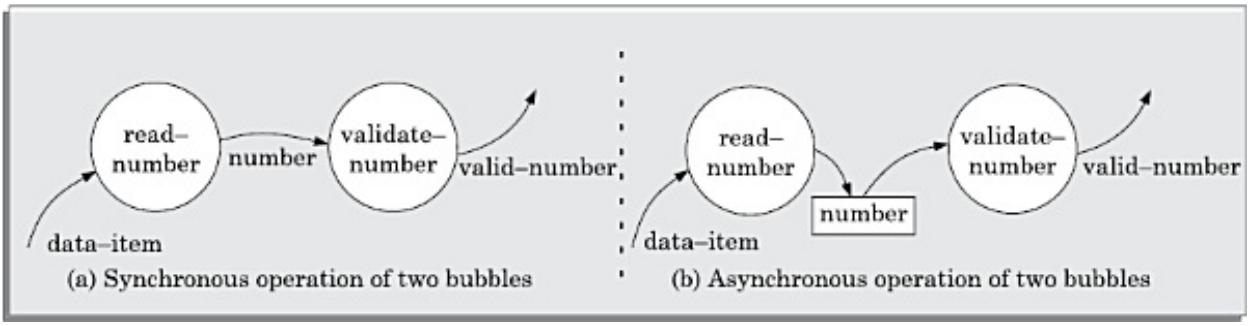

Figure 6.3: Synchronous and asynchronous data flow.

Data dictionary

Every DFD model of a system must be accompanied by a data dictionary. A data dictionary lists all data items that appear in a DFD model. The data items listed include all data flows and the contents of all data stores appearing on all the DFDs in a DFD model. Please remember that the DFD model of a system typically consists of several DFDs, viz., level 0 DFD, level 1 DFD, level 2 DFDs, etc., as shown in Figure 6.4 discussed in new subsection. However, a single data dictionary should capture all the data appearing in all the DFDs constituting the DFD model of a system.

A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items.

For example, a data dictionary entry may represent that the data grossPay consists of the components regularPay and overtimePay.

$$grossPay = regularPay + overtimePay$$

For the smallest units of data items, the data dictionary simply lists their name and their type. Composite data items are expressed in terms of the component data items using certain operators. The operators using which a composite data item can be expressed in terms of its component data items are discussed subsequently.

The dictionary plays a very important role in any software development process, especially for the following reasons:

- A data dictionary provides a standard terminology for all relevant data for use by the developers working in a project. A consistent vocabulary for data items is very important, since in large projects different developers of the project have a tendency to use different terms to refer to the same data, which unnecessarily causes confusion.

- The data dictionary helps the developers to determine the definition of different data structures in terms of their component elements while implementing the design.
- The data dictionary helps to perform impact analysis. That is, it is possible to determine the effect of some data on various processing activities and vice versa. Such impact analysis is especially useful when one wants to check the impact of changing an input value type, or a bug in some functionality, etc.

For large systems, the data dictionary can become extremely complex and voluminous. Even moderate-sized projects can have thousands of entries in the data dictionary. It becomes extremely difficult to maintain a voluminous dictionary manually. Computer-aided software engineering (CASE) tools come handy to overcome this problem. Most CASE tools usually capture the data items appearing in a DFD as the DFD is drawn, and automatically generate the data dictionary. As a result, the designers do not have to spend almost any effort in creating the data dictionary. These CASE tools also support some query language facility to query about the definition and usage of data items. For example, queries may be formulated to determine which data item affects which processes, or a process affects which data items, or the definition and usage of specific data items, etc. Query handling is facilitated by storing the data dictionary in a relational database management system (RDBMS).

Data definition

Composite data items can be defined in terms of primitive data items using the following data definition operators.

+: denotes composition of two data items, e.g. a+b represents data a and b.

[,,]: represents selection, i.e. any one of the data items listed inside the square bracket can occur For example, [a,b] represents either a occurs or b occurs.

(): the contents inside the bracket represent optional data which may or may not appear.

a+(b) represents either a or a+b occurs.

{}: represents iterative data definition, e.g. {name}5 represents five name data. {name}* represents zero or more instances of name data.

=: represents equivalence, e.g. a=b+c means that a is a composite data item

comprising of both b and c.

/* */: Anything appearing within /* and */ is considered as comment.

6.3 DEVELOPING THE DFD MODEL OF A SYSTEM

A DFD model of a system graphically represents how each input data is transformed to its corresponding output data through a hierarchy of DFDs.

The DFD model of a problem consists of many of DFDs and a single data dictionary.

The DFD model of a system is constructed by using a hierarchy of DFDs (see Figure 6.4). The top level DFD is called the level 0 DFD or the context diagram. This is the most abstract (simplest) representation of the system (highest level). It is the easiest to draw and understand. At each successive lower level DFDs, more and more details are gradually introduced. To develop a higher-level DFD model, processes are decomposed into their subprocesses and the data flow among these subprocesses are identified.

To develop the data flow model of a system, first the most abstract representation (highest level) of the problem is to be worked out. Subsequently, the lower level DFDs are developed. Level 0 and Level 1 consist of only one DFD each. Level 2 may contain up to 7 separate DFDs, and level 3 up to 49 DFDs, and so on. However, there is only a single data dictionary for the entire DFD model. All the data names appearing in all DFDs are populated in the data dictionary and the data dictionary contains the definitions of all the data items.

6.3.1 Context Diagram

The context diagram is the most abstract (highest level) data flow representation of a system. It represents the entire system as a single bubble. The bubble in the context diagram is annotated with the name of the software system being developed (usually a noun). This is the only bubble in a DFD model, where a noun is used for naming the bubble. The bubbles at all other levels are annotated with verbs according to the main function performed by the bubble. This is expected since the purpose of the context diagram is to capture the context of the system rather than its functionality. As an example of a context diagram, consider the context diagram a software developed to automate the book keeping activities of a supermarket (see Figure 6.10). The context diagram has been labelled as 'Supermarket software'.

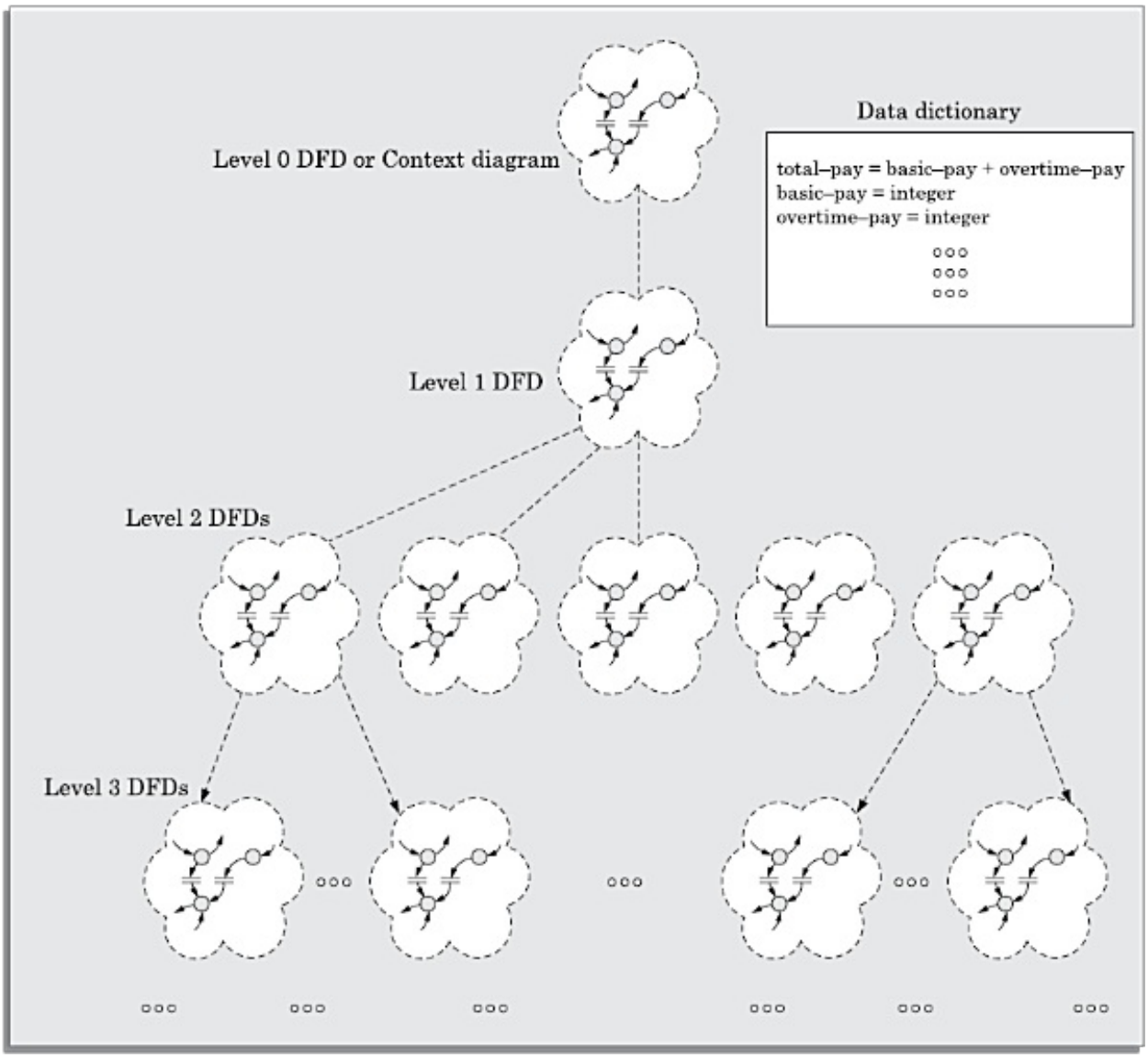

Figure 6.4: DFD model of a system consists of a hierarchy of DFDs and a single data dictionary.

The context diagram establishes the context in which the system operates; that is, who are the users, what data do they input to the system, and what data they received by the system.

The name context diagram of the level 0 DFD is justified because it represents the context in which the system would exist; that is, the external entities who would interact with the system and the specific data items that they would be supplying the system and the data items they would be receiving from the system. The various external entities with which the system interacts and the data flow occurring between the system and the external entities are represented. The data input to the system and the data output from the system are represented as incoming and outgoing arrows. These data flow arrows should be annotated with the corresponding data

names.

To develop the context diagram of the system, we have to analyse the SRS document to identify the different types of users who would be using the system and the kinds of data they would be inputting to the system and the data they would be receiving from the system. Here, the term users of the system also includes any external systems which supply data to or receive data from the system.

6.3.2 Level 1 DFD

The level 1 DFD usually contains three to seven bubbles. That is, the system is represented as performing three to seven important functions. To develop the level 1 DFD, examine the high-level functional requirements in the SRS document. If there are three to seven high-level functional requirements, then each of these can be directly represented as a bubble in the level 1 DFD. Next, examine the input data to these functions and the data output by these functions as documented in the SRS document and represent them appropriately in the diagram.

What if a system has more than seven high-level requirements identified in the SRS document? In this case, some of the related requirements have to be combined and represented as a single bubble in the level 1 DFD. These can be split appropriately in the lower DFD levels. If a system has less than three high-level functional requirements, then some of the high-level requirements need to be split into their subfunctions so that we have roughly about five to seven bubbles represented on the diagram. We illustrate construction of level 1 DFDs in Examples 6.1 to 6.4.

Decomposition

Each bubble in the DFD represents a function performed by the system. The bubbles are decomposed into subfunctions at the successive levels of the DFD model. Decomposition of a bubble is also known as factoring or exploding a bubble. Each bubble at any level of DFD is usually decomposed to anything three to seven bubbles. A few bubbles at any level make that level superfluous. For example, if a bubble is decomposed to just one bubble or two bubbles, then this decomposition becomes trivial and redundant. On the other hand, too many bubbles (i.e. more than seven bubbles) at any level of a DFD makes the DFD model hard to understand. Decomposition of a bubble should be carried

on until a level is reached at which the function of the bubble can be described using a simple algorithm.

We can now describe how to go about developing the DFD model of a system more systematically.

1. **Construction of context diagram:** Examine the SRS document to determine:
   - Different high-level functions that the system needs to perform.
   - Data input to every high-level function.
   - Data output from every high-level function.
   - Interactions (data flow) among the identified high-level functions.

   Represent these aspects of the high-level functions in a diagrammatic form. This would form the top-level data flow diagram (DFD), usually called the DFD 0.

   Construction of level 1 diagram: Examine the high-level functions described in the SRS document. If there are three to seven high-level requirements in the SRS document, then represent each of the high-level function in the form of a bubble. If there are more than seven bubbles, then some of them have to be combined. If there are less than three bubbles, then some of these have to be split.

   Construction of lower-level diagrams: Decompose each high-level function into its constituent subfunctions through the following set of activities:
   - Identify the different subfunctions of the high-level function.
   - Identify the data input to each of these subfunctions.
   - Identify the data output from each of these subfunctions.
   - Identify the interactions (data flow) among these subfunctions.

   Represent these aspects in a diagrammatic form using a DFD.

   Recursively repeat Step 3 for each subfunction until a subfunction can be represented by using a simple algorithm.

Numbering of bubbles

It is necessary to number the different bubbles occurring in the DFD. These numbers help in uniquely identifying any bubble in the DFD from its bubble number. The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD. Bubbles at level 1 are numbered, 0.1, 0.2, 0.3, etc. When a bubble numbered x is

decomposed, its children bubble are numbered x.1, x.2, x.3, etc. In this numbering scheme, by looking at the number of a bubble we can unambiguously determine its level, its ancestors, and its successors.

Balancing DFDs

The DFD model of a system usually consists of many DFDs that are organised in a hierarchy. In this context, a DFD is required to be balanced with respect to the corresponding bubble of the parent DFD.

> The data that flow into or out of a bubble must match the data flow at the next level of DFD. This is known as balancing a DFD.

We illustrate the concept of balancing a DFD in Figure 6.5. In the level 1 DFD, data items d1 and d3 flow out of the bubble 0.1 and the data item d2 flows into the bubble 0.1 (shown by the dotted circle). In the next level, bubble 0.1 is decomposed into three DFDs (0.1.1,0.1.2,0.1.3). The decomposition is balanced, as d1 and d3 flow out of the level 2 diagram and d2 flows in. Please note that dangling arrows (d1,d2,d3) represent the data flows into or out of a diagram.

How far to decompose?

A bubble should not be decomposed any further once a bubble is found to represent a simple set of instructions. For simple problems, decomposition up to level 1 should suffice. However, large industry standard problems may need decomposition up to level 3 or level 4. Rarely, if ever, decomposition beyond level 4 is needed.

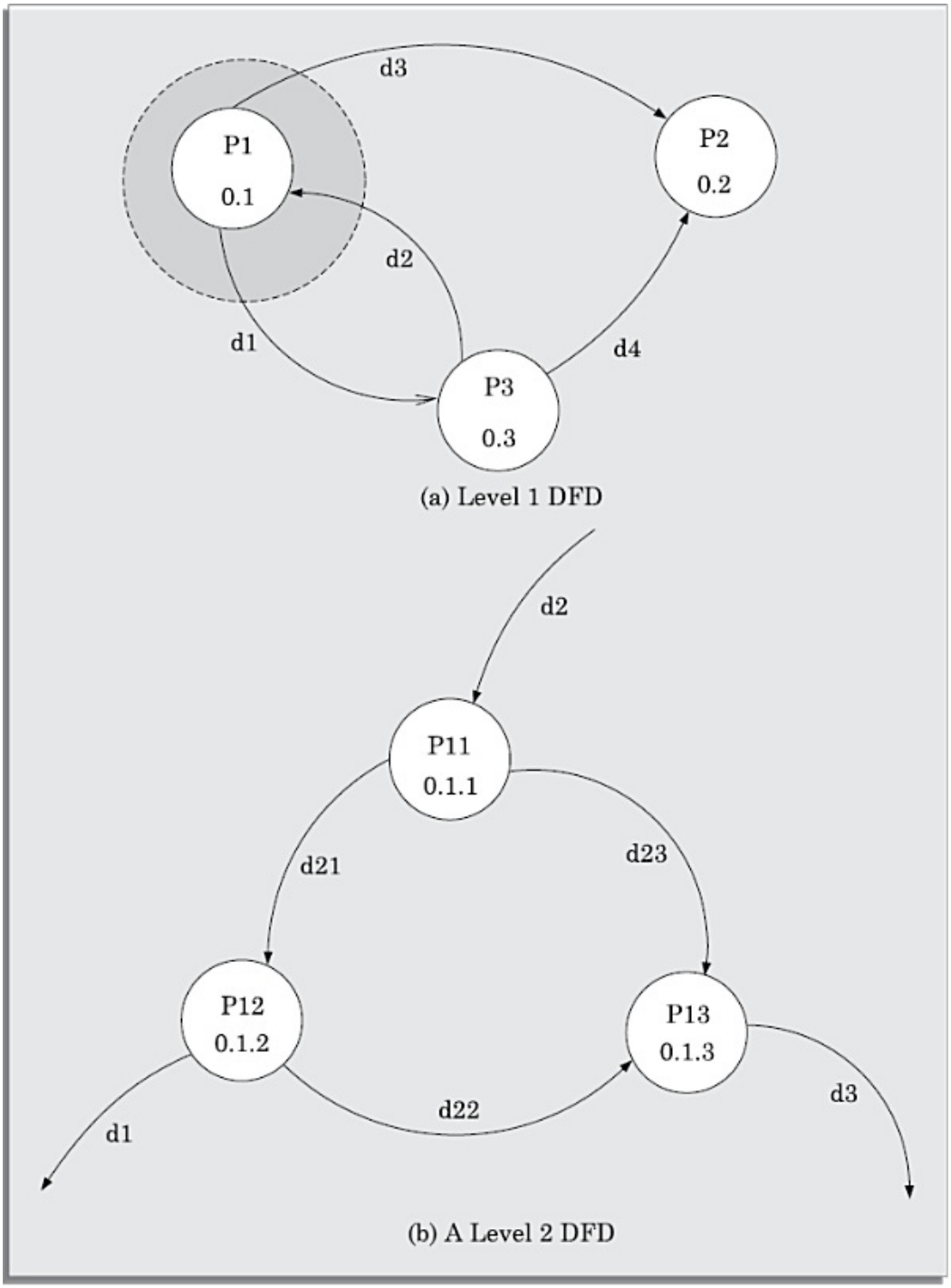

Figure 6.5: An example showing balanced decomposition.

Commonly made errors while constructing a DFD model

Although DFDs are simple to understand and draw, students and practitioners alike encounter similar types of problems while modelling software problems using DFDs. While learning from experience is a powerful thing, it is an expensive pedagogical technique in the business world. It is therefore useful to understand the different types of mistakes that beginners usually make while constructing the DFD model

of systems, so that you can consciously try to avoid them.The errors are as follows:

- Many beginners commit the mistake of drawing more than one bubble in the context diagram. Context diagram should depict the system as a single bubble.
- Many beginners create DFD models in which external entities appearing at all levels of DFDs. All external entities interacting with the system should be represented only in the context diagram. The external entities should not appear in the DFDs at any other level.
- It is a common oversight to have either too few or too many bubbles in a DFD. Only three to seven bubbles per diagram should be allowed. This also means that each bubble in a DFD should be decomposed three to seven bubbles in the next level.
- Many beginners leave the DFDs at the different levels of a DFD model unbalanced.
- A common mistake committed by many beginners while developing a DFD model is attempting to represent control information in a DFD.

It is important to realise that a DFD represents only data flow, and it does not represent any control information.

The following are some illustrative mistakes of trying to represent control aspects such as:

Illustration 1. A book can be searched in the library catalog by inputting its name. If the book is available in the library, then the details of the book are displayed. If the book is not listed in the catalog, then an error message is generated. While developing the DFD model for this simple problem, many beginners commit the mistake of drawing an arrow (as shown in Figure 6.6) to indicate that the error function is invoked after the search book. But, this is a control information and should not be shown on the DFD.

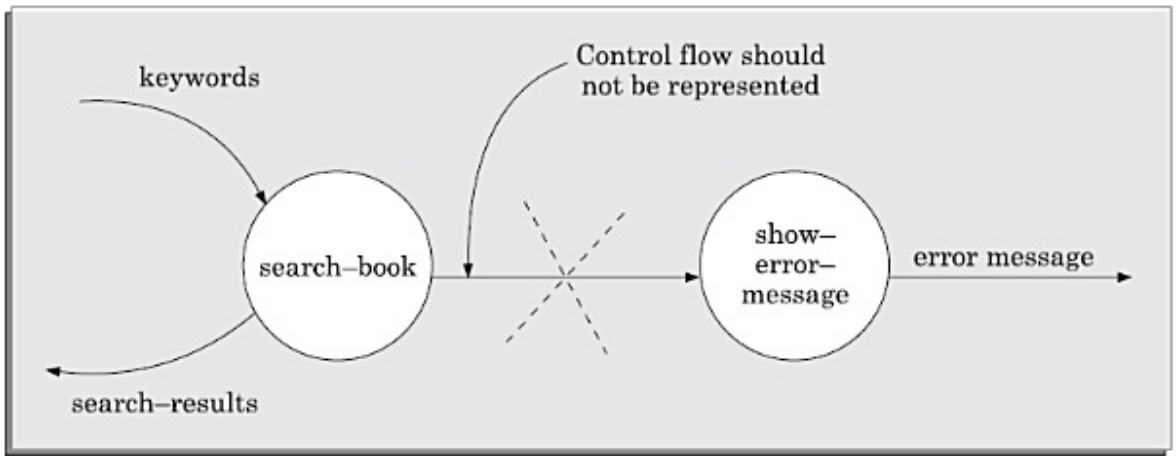

Figure 6.6: It is incorrect to show control information on a DFD.

Illustration 2. Another type of error occurs when one tries to represent when or in what order different functions (processes) are invoked. A DFD similarly should not represent the conditions under which different functions are invoked.

Illustration 3. If a bubble A invokes either the bubble B or the bubble C depending upon some conditions, we need only to represent the data that flows between bubbles A and B or bubbles A and C and not the conditions depending on which the two modules are invoked.

- A data flow arrow should not connect two data stores or even a data store with an external entity. Thus, data cannot flow from a data store to another data store or to an external entity without any intervening processing. As a result, a data store should be connected only to bubbles through data flow arrows.
- All the functionalities of the system must be captured by the DFD model. No function of the system specified in the SRS document of the system should be overlooked.
- Only those functions of the system specified in the SRS document should be represented. That is, the designer should not assume functionality of the system not specified by the SRS document and then try to represent them in the DFD.
- Incomplete data dictionary and data dictionary showing incorrect composition of data items are other frequently committed mistakes.
- The data and function names must be intuitive. Some students and even practicing developers use meaningless symbolic data names such as a,b,c, etc. Such names hinder understanding the DFD model.

- Novices usually clutter their DFDs with too many data flow arrow. It becomes difficult to understand a DFD if any bubble is associated with more than seven data flows. When there are too many data flowing in or out of a DFD, it is better to combine these data items into a high-level data item. Figure 6.7 shows an example concerning how a DFD can be simplified by combining several data flows into a single high-level data flow.

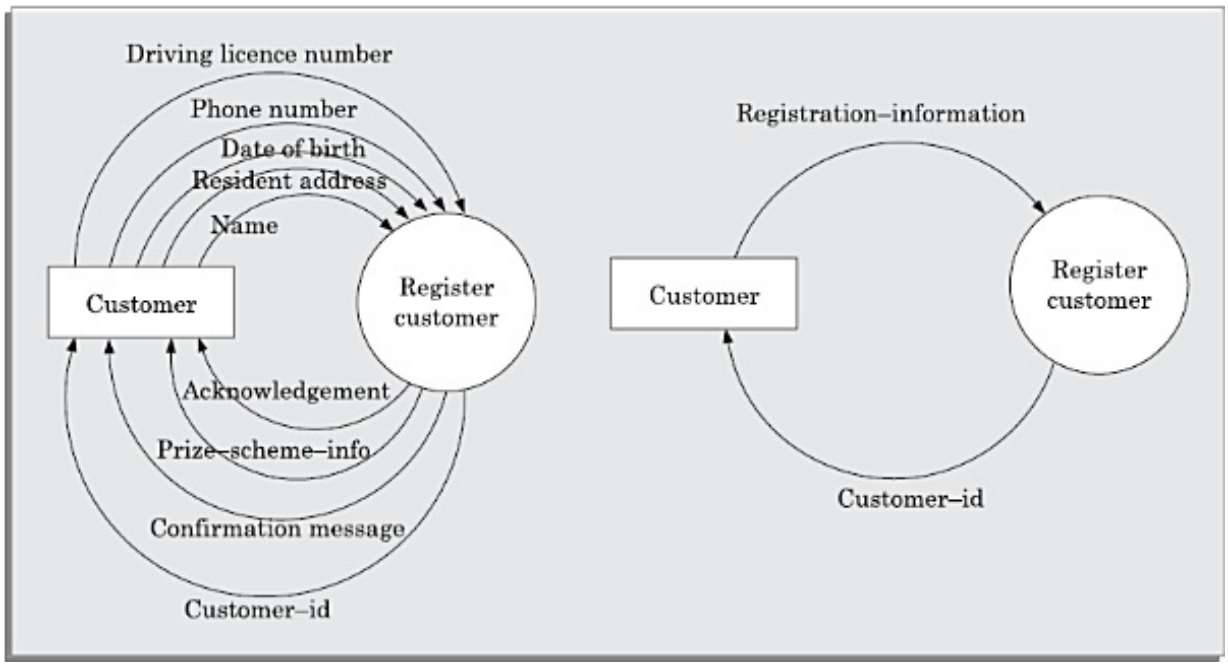

Figure 6.7: Illustration of how to avoid data cluttering.

We now illustrate the structured analysis technique through a few examples.

Example 6.1 (RMS Calculating Software) A software system called RMS calculating software would read three integral numbers from the user in the range of –1000 and +1000 and would determine the root mean square (RMS) of the three input numbers and display it.

In this example, the context diagram is simple to draw. The system accepts three integers from the user and returns the result to him. This has been shown in Figure 6.8(a). To draw the level 1 DFD, from a cursory analysis of the problem description, we can see that there are four basic functions that the system needs to perform—accept the input numbers from the user, validate the numbers, calculate the root mean square of the input numbers and, then display the result. After representing these four functions in Figure 6.8(b), we observe that the calculation of root mean square essentially consists of the functions—calculate the squares of the input numbers,

calculate the mean, and finally calculate the root. This decomposition is shown in the level 2 DFD in Figure 6.8(c).

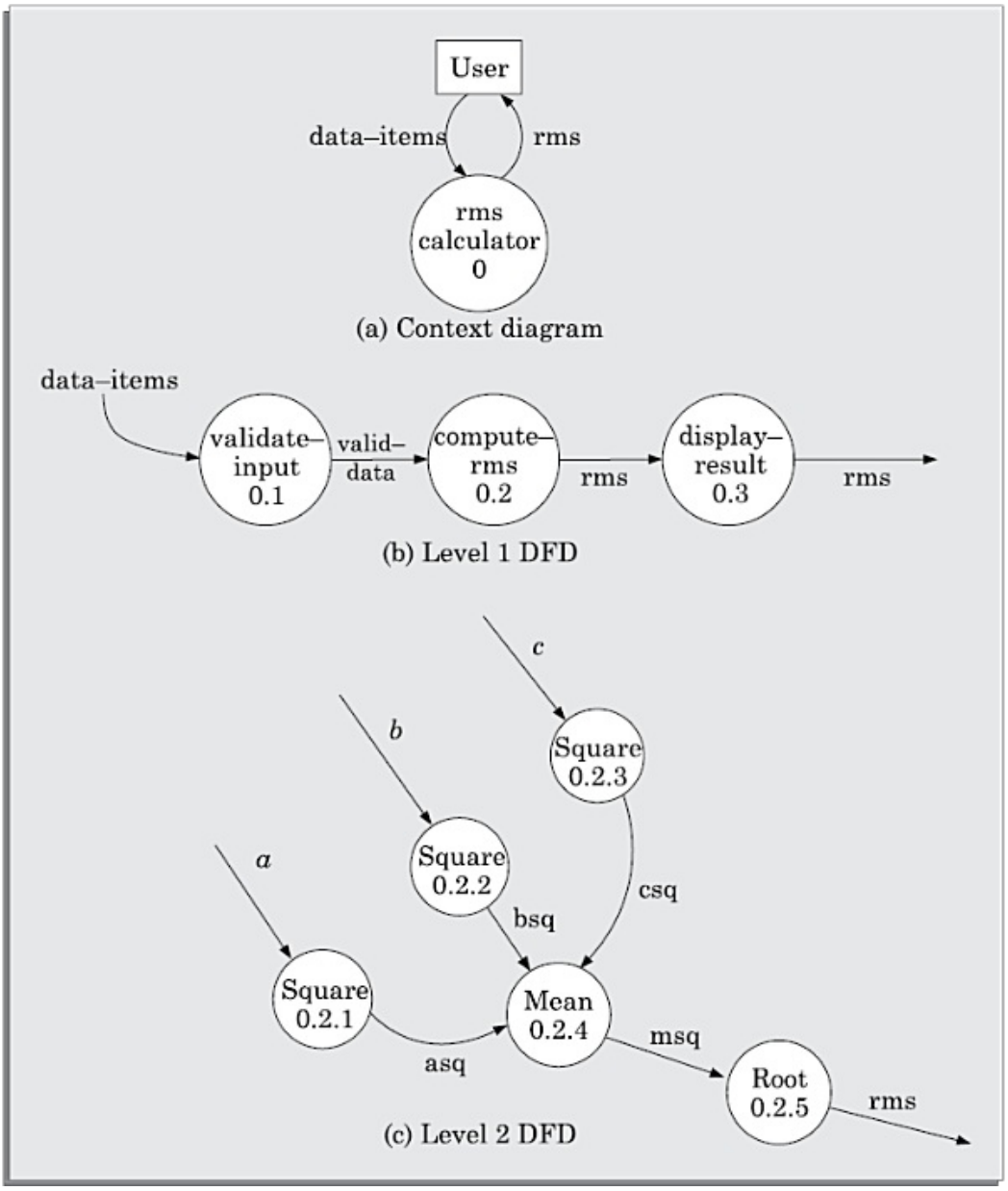

Figure 6.8: Context diagram, level 1, and level 2 DFDs for Example 6.1.

Data dictionary for the DFD model of Example 6.1

data-items: {integer}3
rms: float
valid-data:data-items
a: integer
b: integer
c: integer
asq: integer

bsq: integer
csq: integer
msq: integer

Example 6.1 is an almost trivial example and is only meant to illustrate the basic methodology. Now, let us perform the structured analysis for a more complex problem.

Example 6.2 (Tic-Tac-Toe Computer Game) Tic-tac-toe is a computer game in which a human player and the computer make alternate moves on a 3×3 square. A move consists of marking a previously unmarked square. The player who is first to place three consecutive marks along a straight line (i.e., along a row, column, or diagonal) on the square wins. As soon as either of the human player or the computer wins, a message congratulating the winner should be displayed. If neither player manages to get three consecutive marks along a straight line, and all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game.

The context diagram and the level 1 DFD are shown in Figure 6.9.

Data dictionary for the DFD model of Example 6.2

move: integer /* number between 1 to 9 */
display: game+result
game: board
board: {integer}9
result: ["computer won", "human won", "drawn"]

Example 6.3 (Supermarket Prize Scheme) A super market needs to develop a software that would help it to automate a scheme that it plans to introduce to encourage regular customers. In this scheme, a customer would have first register by supplying his/her residence address, telephone number, and the driving license number. Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer. A customer can present his CN to the check out staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year. Also, it intends to award a 22 caret gold coin to every customer whose purchase exceeded Rs. 10,000. The entries against the CN are reset on the last day of every year after the prize winners' lists are generated.

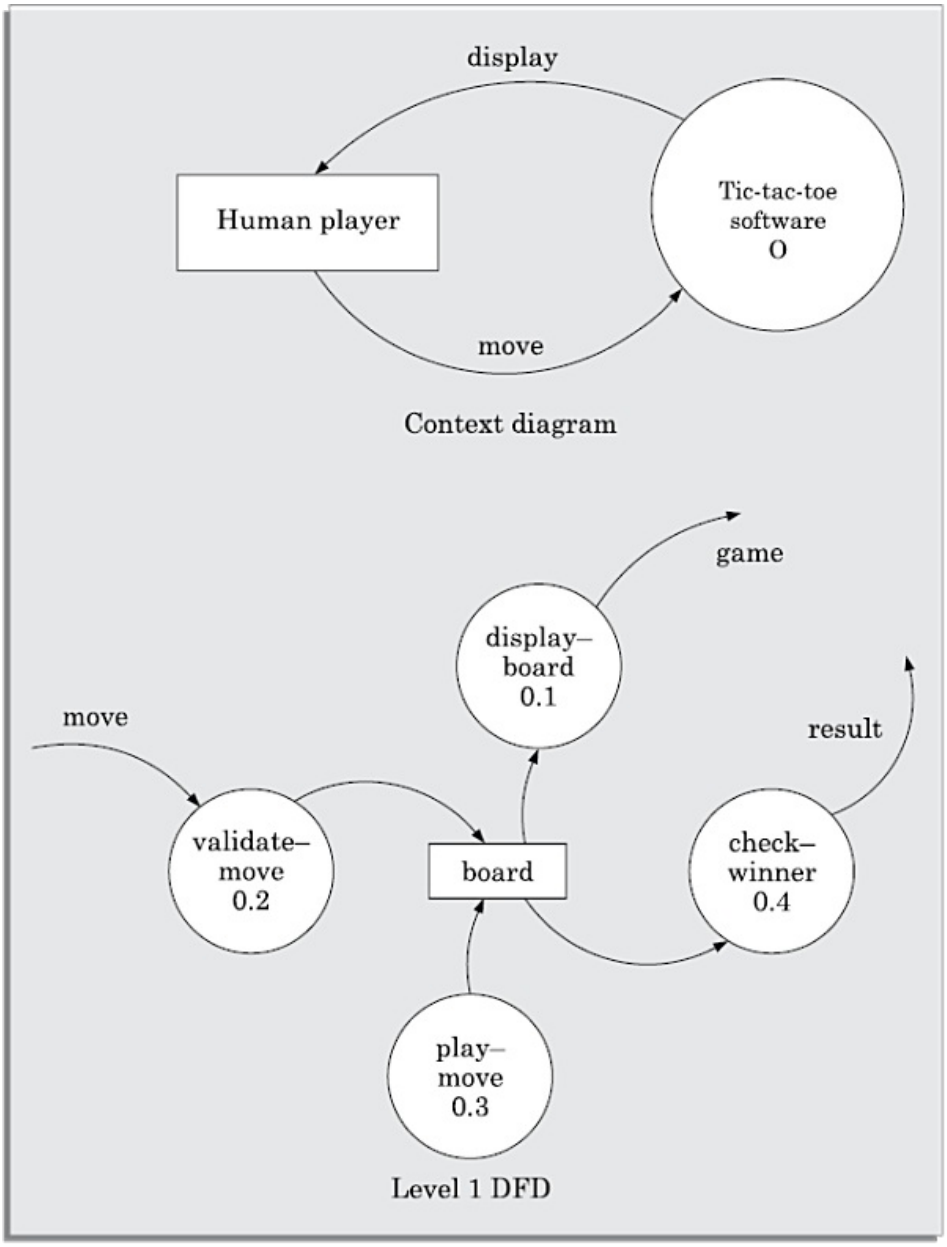

Figure 6.9: Context diagram and level 1 DFDs for Example 6.2.

The context diagram for the supermarket prize scheme problem of Example 6.3 is shown in Figure 6.10. The level 1 DFD in Figure 6.11. The level 2 DFD in Figure 6.12.

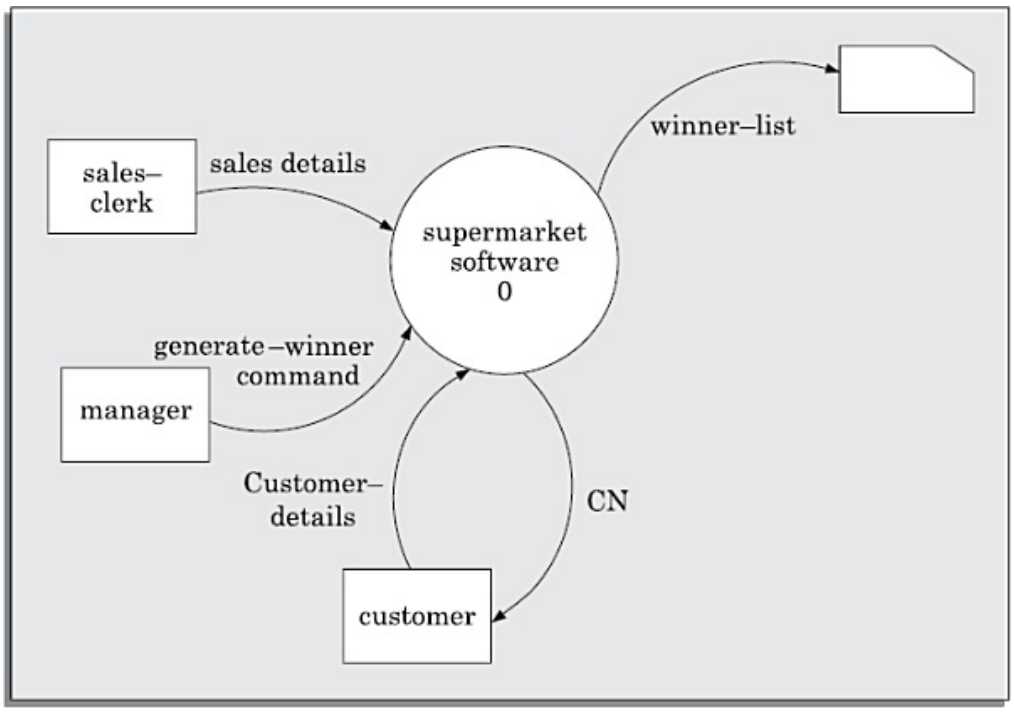

Figure 6.10: Context diagram for Example 6.3.

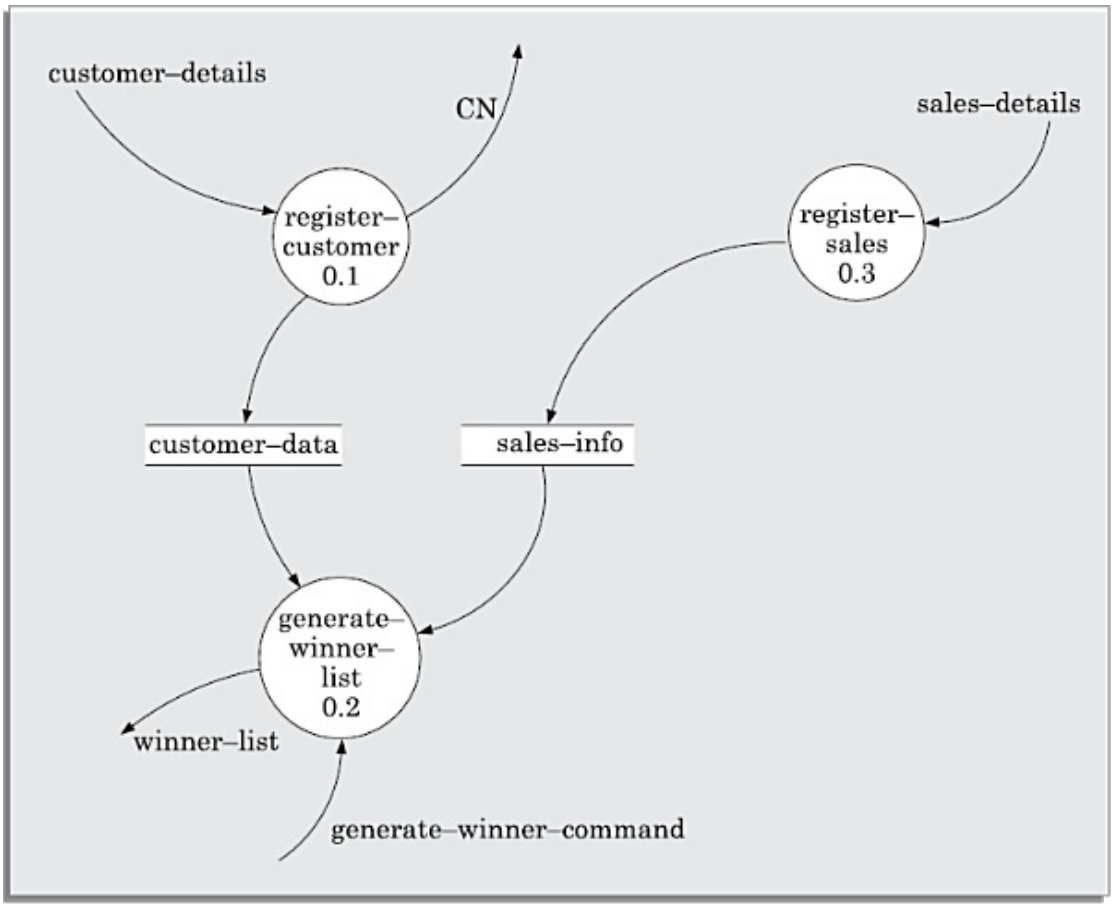

Figure 6.11: Level 1 diagram for Example 6.3.

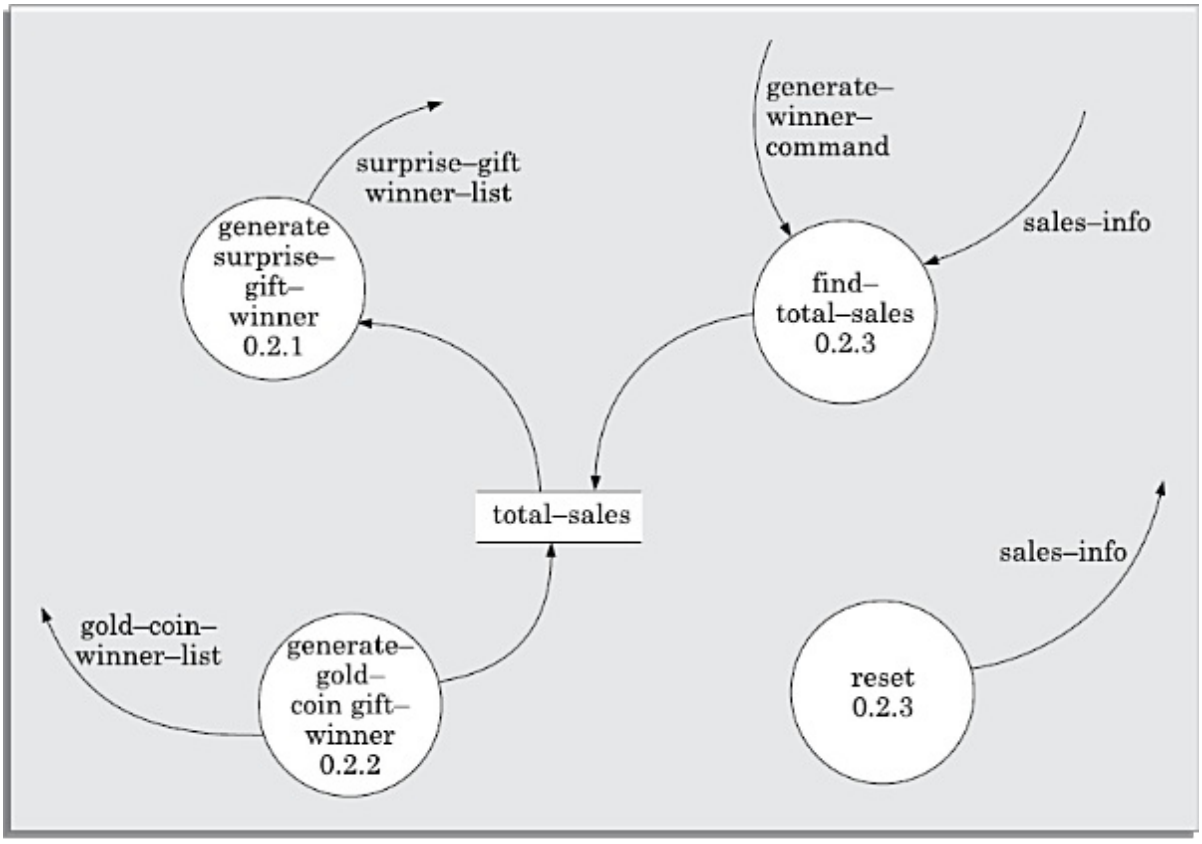

Figure 6.12: Level 2 diagram for Example 6.3.

Data dictionary for the DFD model of Example 6.3

address: name+house#+street#+city+pin
sales-details: {item+amount}* + CN
CN: integer
customer-data: {address+CN}*
sales-info: {sales-details}*
winner-list: surprise-gift-winner-list + gold-coin-winner-list
surprise-gift-winner-list: {address+CN}*
gold-coin-winner-list: {address+CN}*
gen-winner-command: command
total-sales: {CN+integer}*

Observations: The following observations can be made from the Example 6.3.

1. The fact that the customer is issued a manually prepared customer identity card or that the customer hands over the identity card each time he makes a purchase has not been shown in the DFD. This is because these are item transfers occurring outside the computer.
2. The data generate-winner-list in a way represents control information

(that is, command to the software) and no real data. We have included it in the DFD because it simplifies the structured design process as we shall realize after we practise solving a few problems. We could have also as well done without the generate-winner-list data, but this could have a bit complicated the design.

3. Observe in Figure 6.11 that we have two separate stores for the customer data and sales data. Should we have combined them into a single data store? The answer is—No, we should not. If we had combined them into a single data store, the structured design that would be carried out based on this model would become complicated. Customer data and sales data have very different characteristics. For example, customer data once created, does not change. On the other hand, the sales data changes frequently and also the sales data is reset at the end of a year, whereas the customer data is not.

Example 6.4 (Trading-house Automation System (TAS)) A trading house wants us to develop a computerized system that would automate various book-keeping activities associated with its business. The following are the salient features of the system to be developed:

- The trading house has a set of regular customers. The customers place orders with it for various kinds of commodities. The trading house maintains the names and addresses of its regular customers. Each of these regular customers should be assigned a unique customer identification number (CIN) by the computer. The customers quote their CIN on every order they place.
- Once order is placed, as per current practice, the accounts department of the trading house first checks the credit-worthiness of the customer. The credit-worthiness of the customer is determined by analysing the history of his payments to different bills sent to him in the past. After automation, this task has be done by the computer.
- If a customer is not credit-worthy, his orders are not processed any further and an appropriate order rejection message is generated for the customer.
- If a customer is credit-worthy, the items that he has ordered are checked against the list of items that the trading house deals with. The items in the order which the trading house does not deal with, are not processed any further and an appropriate apology message for the

customer for these items is generated.

- The items in the customer's order that the trading house deals with are checked for availability in the inventory. If the items are available in the inventory in desired quantity, then:

– A bill is with the forwarding address of the customer is printed.
– A material issue slip is printed. The customer can produce this material issue slip at the store house and take delivery of the items.
– Inventory data is adjusted to reflect the sale to the customer.

- If any of the ordered items are not available in the inventory in sufficient quantity to satisfy the order, then these out-of-stock items along with the quantity ordered by the customer and the CIN are stored in a "pending-order" file for further processing to be carried out when the purchase department issues the "generate indent" command.
- The purchase department should be allowed to periodically issue commands to generate indents. When a command to generate indents is issued, the system should examine the "pending-order" file to determine the orders that are pending and determine the total quantity required for each of the items. It should find out the addresses of the vendors who supply these items by examining a file containing vendor details and then should print out indents to these vendors.
- The system should also answer managerial queries regarding the statistics of different items sold over any given period of time and the corresponding quantity sold and the price realised.

The context diagram for the trading house automation problem is shown in Figure 6.13. The level 1 DFD in Figure 6.14.

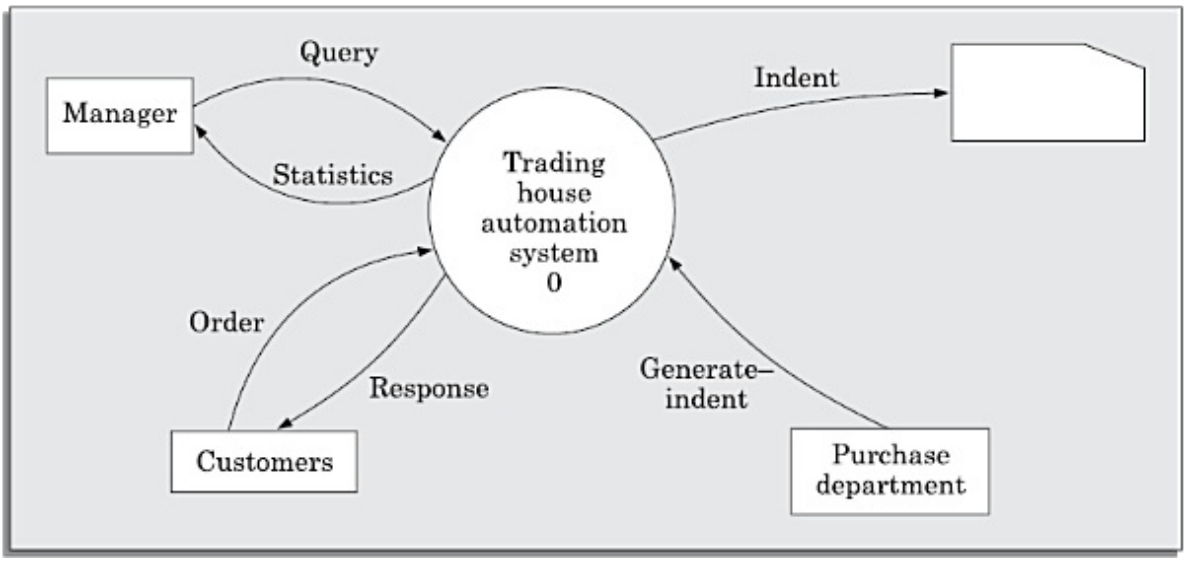

Figure 6.13: Context diagram for Example 6.4.

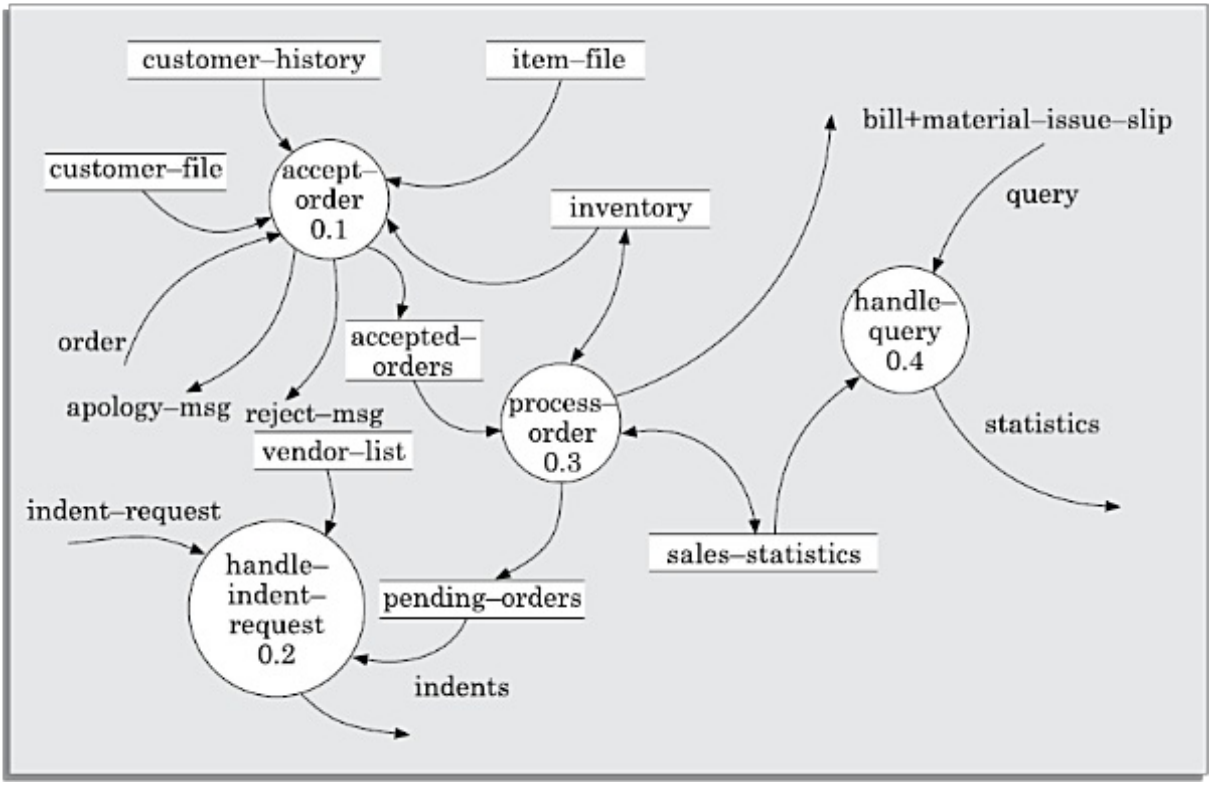

Figure 6.14: Level 1 DFD for Example 6.4.

Data dictionary for the DFD model of Example 6.4

response: [bill + material-issue-slip, reject-msg,apology-msg]
query: period /* query from manager regarding sales statistics*/
period: [date+date,month,year,day]
date: year + month + day year: integer
month: integer day: integer customer-id: integer
order: customer-id + {items + quantity}* + order#

accepted-order: order /* ordered items available in inventory */
reject-msg: order + message /* rejection message */
pending-orders: customer-id + order# + {items+quantity}*
customer-address: name+house#+street#+city+pin
name: string
house#: string
street#: string
city: string
pin: integer
customer-id: integer
customer-file: {customer-address}* + customer-id
bill: {item + quantity + price}* + total-amount + customer-address + order#
material-issue-slip: message + item + quantity + customer-address
message: string
statistics: {item + quantity + price }*
sales-statistics: {statistics}* + date
quantity: integer
order#: integer /* unique order number generated by the program */
price: integer
total-amount: integer
generate-indent: command
indent: {item+quantity}* + vendor-address
indents: {indent}*
vendor-address: customer-address
vendor-list: {vendor-address}*
item-file: {item}*
item: string
indent-request: command

Observations: The following observations can be made from Example 6.4.

1. In a DFD, if two data stores deal with different types of data, e.g. one type of data is invariant with time whereas another varies with time, (e.g. vendor address, and inventory data) it is a good idea to represent them as separate data stores.

If two types of data always get updated at the same time, they should be stored in a single data store. Otherwise, separate data stores should be used for them.

The inventory data changes each time supply arrives and the inventory

is updated or an item is sold, whereas the vendor data remains unchanged.

2. If we are developing the DFD model of a process which is already being manually carried out, then the names of the registers being maintained in the manual process would appear as data stores in the DFD model. For example, if TAS is currently being manually carried out, then normally there would registers corresponding to accepted orders, pending orders, vendor list, etc.
3. We can observe that DFDs enable a software developer to develop the data domain and functional domain model of the system at the same time. As the DFD is refined into greater levels of detail, the analyst performs an implicit functional decomposition. At the same time, the DFD refinement automatically results in refinement of corresponding data items.
4. The data that are maintained in physical registers in manual processing, become data stores in the DFD representation. Therefore, to determine which data should be represented as a data store, it is useful to try to imagine whether a set of data items would be maintained in a register in a manual system.

Example 6.5 (Personal Library Software) Perform structured analysis for the personal library software of Example 6.5.

The context diagram is shown in Figure 6.15.

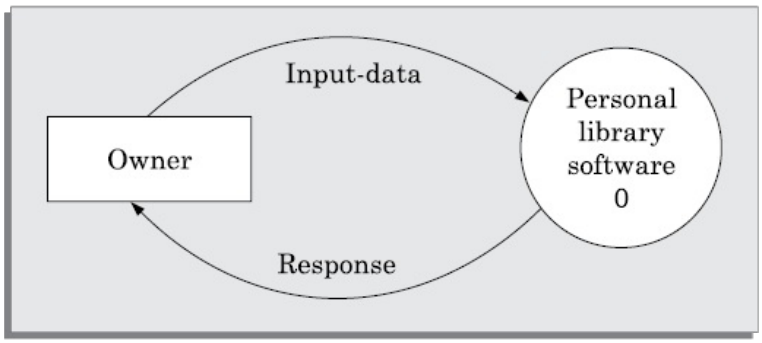

Figure 6.15: Context diagram for Example 6.5.

The level 1 DFD is shown in Figure 6.16.

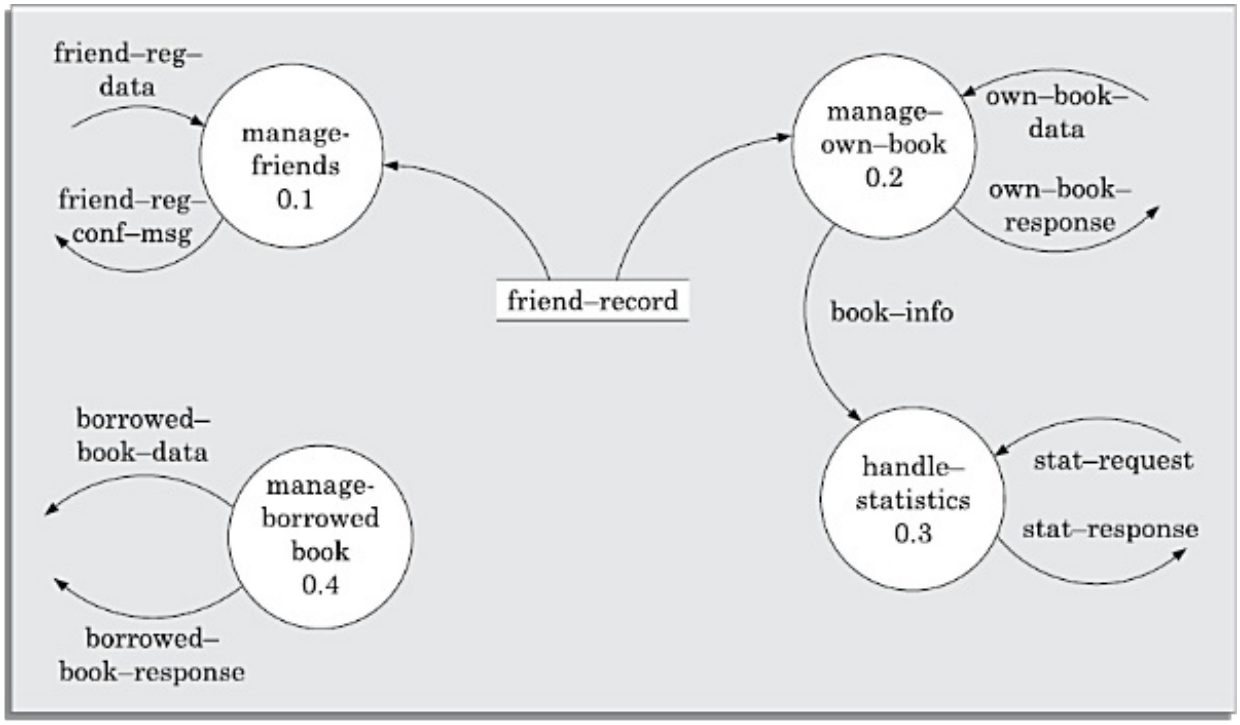

Figure 6.16: Level 1 DFD for Example 6.5.

The level 2 DFD for the manageOwnBook bubble is shown in Figure 6.17.

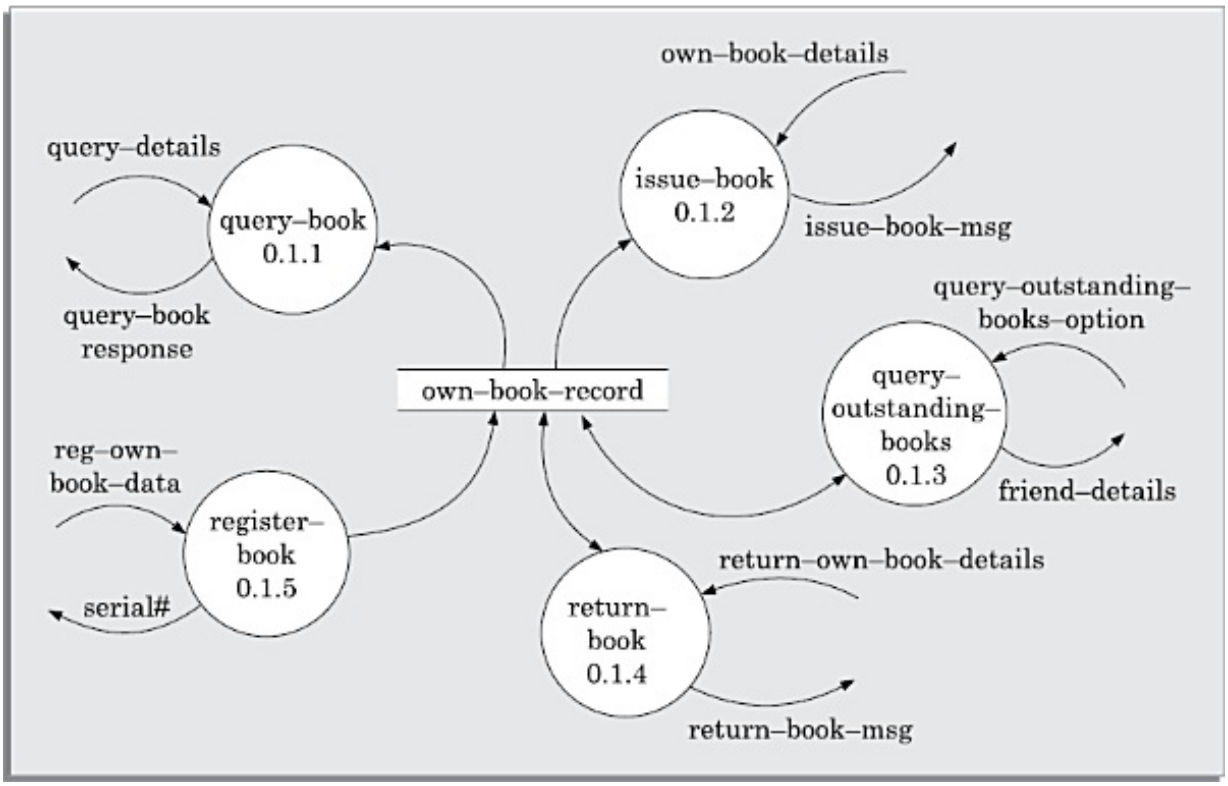

Figure 6.17: Level 2 DFD for Example 6.5.

Data dictionary for the DFD model of Example 6.5

input-data: friend-reg-data + own-book-data + stat-request + borrowed-book-data

response: friend-reg-conf-msg + own-book-response + stat-response + borrowed-book-response
own-book-data: query-details + own-book-details + query-outstanding-books-option + return-own book-details + reg-own-book-data
own-book-response: query-book-response + issue-book-msg + friend-details + return-book- msg + serial#.
borrowed-book-data: borrowed-book-details + book-return-details + display-books-option borrowed-book-response: reg-msg + unreg-msg + borrowed-books-list
friend-reg-data: name + address + landline# + mobile#
own-book-details: friend-reg-data + book-title + data-of-issue
return-own-book-details: book-title + date-of-return
friend-details: name + address + landline# + mobile# + book-list
borrowed-book-details: book-title + borrow-date
serial#: integer

Observation: Observe that since there are more than seven functional requirements for the personal library software, related requirements have been combined to have only five bubbles in the level 1 diagram. Only level 2 DFD has been shown, since the other DFDs are trivial and need not be drawn.

Shortcomings of the DFD model

DFD models suffer from several shortcomings. The important shortcomings of DFD models are the following:

- Imprecise DFDs leave ample scope to be imprecise. In the DFD model, we judge the function performed by a bubble from its label. However, a short label may not capture the entire functionality of a bubble. For example, a bubble named find-book-position has only intuitive meaning and does not specify several things, e.g. what happens when some input information is missing or is incorrect. Further, the find-book-position bubble may not convey anything regarding what happens when the required book is missing.
- Not-well defined control aspects are not defined by a DFD. For instance, the order in which inputs are consumed and outputs are produced by a bubble is not specified. A DFD model does not specify the order in which the different bubbles are executed. Representation of such aspects is very important for modelling real-time systems.
- Decomposition: The method of carrying out decomposition to arrive at the successive levels and the ultimate level to which decomposition is carried out are highly subjective and depend on the choice and judgment of the analyst. Due to this reason, even for the same problem, several alternative DFD representations are possible. Further, many times it is not possible to say which DFD representation is

superior or preferable to another one.

- Improper data flow diagram: The data flow diagramming technique does not provide any specific guidance as to how exactly to decompose a given function into its subfunctions and we have to use subjective judgment to carry out decomposition.

6.3.3 Extending DFD Technique to Make it Applicable to Real-time Systems

In a real-time system, some of the high-level functions are associated with deadlines. Therefore, a function must not only produce correct results but also should produce them by some prespecified time. For real-time systems, execution time is an important consideration for arriving at a correct design. Therefore, explicit representation of control and event flow aspects are essential. One of the widely accepted techniques for extending the DFD technique to real-time system analysis is the Ward and Mellor technique [1985]. In the Ward and Mellor notation, a type of process that handles only control flows is introduced. These processes representing control processing are denoted using dashed bubbles. Control flows are shown using dashed lines/arrows.

Unlike Ward and Mellor, Hatley and Pirbhai [1987] show the dashed and solid representations on separate diagrams. To be able to separate the data processing and the control processing aspects, a control flow diagram (CFD) is defined. This reduces the complexity of the diagrams. In order to link the data processing and control processing diagrams, a notational reference (solid bar) to a control specification is used. The CSPEC describes the following:

- The effect of an external event or control signal.
- The processes that are invoked as a consequence of an event.

Control specifications represents the behavior of the system in two different ways:

- It contains a state transition diagram (STD). The STD is a sequential specification of behaviour.
- It contains a program activation table (PAT). The PAT is a combinatorial specification of behaviour. PAT represents invocation

sequence of bubbles in a DFD.

6.4 STRUCTURED DESIGN

The aim of structured design is to transform the results of the structured analysis (that is, the DFD model) into a structure chart. A structure chart represents the software architecture. The various modules making up the system, the module dependency (i.e. which module calls which other modules), and the parameters that are passed among the different modules. The structure chart representation can be easily implemented using some programming language. Since the main focus in a structure chart representation is on module structure of a software and the interaction among the different modules, the procedural aspects (e.g. how a particular functionality is achieved) are not represented.

The basic building blocks using which structure charts are designed are as following:

Rectangular boxes: A rectangular box represents a module. Usually, every rectangular box is annotated with the name of the module it represents.

Module invocation arrows: An arrow connecting two modules implies that during program execution control is passed from one module to the other in the direction of the connecting arrow. However, just by looking at the structure chart, we cannot say whether a modules calls another module just once or many times. Also, just by looking at the structure chart, we cannot tell the order in which the different modules are invoked.

Data flow arrows: These are small arrows appearing alongside the module invocation arrows. The data flow arrows are annotated with the corresponding data name. Data flo w arrows represent the fact that the named data passes from one module to the other in the direction of the arrow.

Library modules: A library module is usually represented by a rectangle with double edges. Libraries comprise the frequently called modules. Usually, when a module is invoked by many other modules, it is made into a library module.

Selection: The diamond symbol represents the fact that one module of several modules connected with the diamond symbol is invoked depending on the outcome of the condition attached with the diamond symbol.

Repetition: A loop around the control flow arrows denotes that the respective modules are invoked repeatedly.

In any structure chart, there should be one and only one module at the top, called the root. There should be at most one control relationship between any two modules in the structure chart. This means that if module A invokes module B, module B cannot invoke module A. The main reason behind this restriction is that we can consider the different modules of a structure chart to be arranged in layers or levels. The principle of abstraction does not allow lower-level modules to be aware of the existence of the high-level modules. However, it is possible for two higher-level modules to invoke the same lower-level module. An example of a properly layered design and another of a poorly layered design are shown in Figure 6.18.

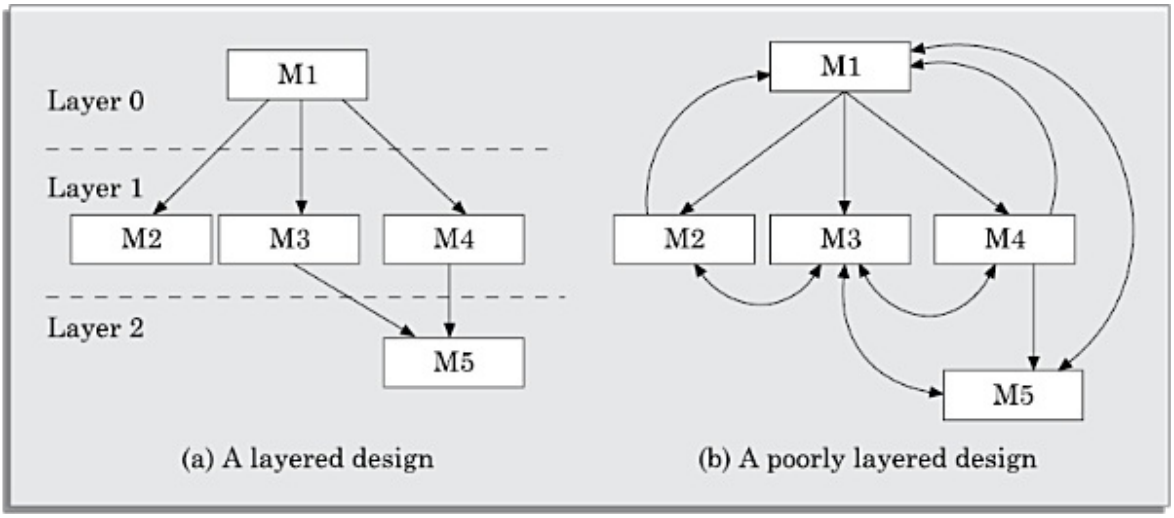

Figure 6.18: Examples of properly and poorly layered designs.

Flow chart *versus* structure chart

We are all familiar with the flow chart representation of a program. Flow chart is a convenient technique to represent the flow of control in a program. A structure chart differs from a flow chart in three principal ways:

- It is usually difficult to identify the different modules of a program from its flow chart representation.
- Data interchange among different modules is not represented in a flow chart.
- Sequential ordering of tasks that is inherent to a flow chart is suppressed in a structure chart.

6.4.1 Transformation of a DFD Model into Structure Chart

Systematic techniques are available to transform the DFD representation of a problem into a module structure represented by as a structure chart. Structured design provides two strategies to guide transformation of a DFD into a structure chart:

- Transform analysis
- Transaction analysis

Normally, one would start with the level 1 DFD, transform it into module representation using either the transform or transaction analysis and then proceed toward the lower level DFDs.

At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD.

Whether to apply transform or transaction processing?

Given a specific DFD of a model, how does one decide whether to apply transform analysis or transaction analysis? For this, one would have to examine the data input to the diagram. The data input to the diagram can be easily spotted because they are represented by dangling arrows. If all the data flow into the diagram are processed in similar ways (i.e. if all the input data flow arrows are incident on the same bubble in the DFD) then transform analysis is applicable. Otherwise, transaction analysis is applicable. Normally, transform analysis is applicable only to very simple processing.

Please recollect that the bubbles are decomposed until it represents a very simple processing that can be implemented using only a few lines of code. Therefore, transform analysis is normally applicable at the lower levels of a DFD model. Each different way in which data is processed corresponds to a separate transaction. Each transaction corresponds to a functionality that lets a user perform a meaningful piece of work using the software.

Transform analysis

Transform analysis identifies the primary functional components (modules) and the input and output data for these components. The first step in transform analysis is to divide the DFD into three types of parts:

- Input.
- Processing.
- Output.

The input portion in the DFD includes processes that transform input data from physical (e.g, character from terminal) to logical form (e.g. internal tables, lists, etc.). Each input portion is called an afferent branch.

The output portion of a DFD transforms output data from logical form to physical form. Each output portion is called an efferent branch. The remaining portion of a DFD is called central transform.

In the next step of transform analysis, the structure chart is derived by drawing one functional component each for the central transform, the afferent and efferent branches. These are drawn below a root module, which would invoke these modules.

Identifying the input and output parts requires experience and skill. One possible approach is to trace the input data until a bubble is found whose output data cannot be deduced from its inputs alone. Processes which validate input are not central transforms. Processes which sort input or filter data from it are central tansforms. The first level of structure chart is produced by representing each input and output unit as a box and each central transform as a single box.

In the third step of transform analysis, the structure chart is refined by adding subfunctions required by each of the high-level functional components. Many levels of functional components may be added. This process of breaking functional components into subcomponents is called factoring. Factoring includes adding read and write modules, error-handling modules, initialisation and termination process, identifying consumer modules etc. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

Example 6.6 Draw the structure chart for the RMS software of Example 6.1.

By observing the level 1 DFD of Figure 6.8, we can identify validate-input as the afferent branch and write-output as the efferent branch. The remaining (i.e., compute-rms) as the central transform. By applying the step 2 and step 3 of transform analysis, we get the structure chart shown in Figure 6.19.

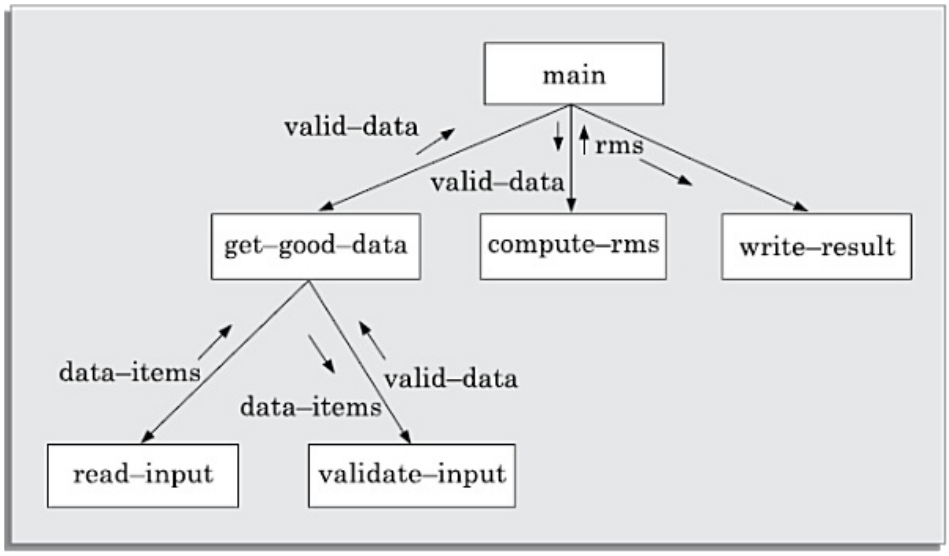

Figure 6.19: Structure chart for Example 6.6.

Example 6.7 Draw the structure chart for the tic-tac-toe software of Example 6.2.

The structure chart for the Tic-tac-toe software is shown in Figure 6.20. Observe that the check-game-status bubble, though produces some outputs. is not really responsible for converting logical data to physical data. On the other hand, it carries out the processing involving checking game status. That is the main reason, why we have considered it as a central transform and not as an efferent type of module.

Transaction analysis

Transaction analysis is an alternative to transform analysis and is useful while designing transaction processing programs. A transaction allows the user to perform some specific type of work by using the software. For example, 'issue book', 'return book', 'query book', etc., are transactions.

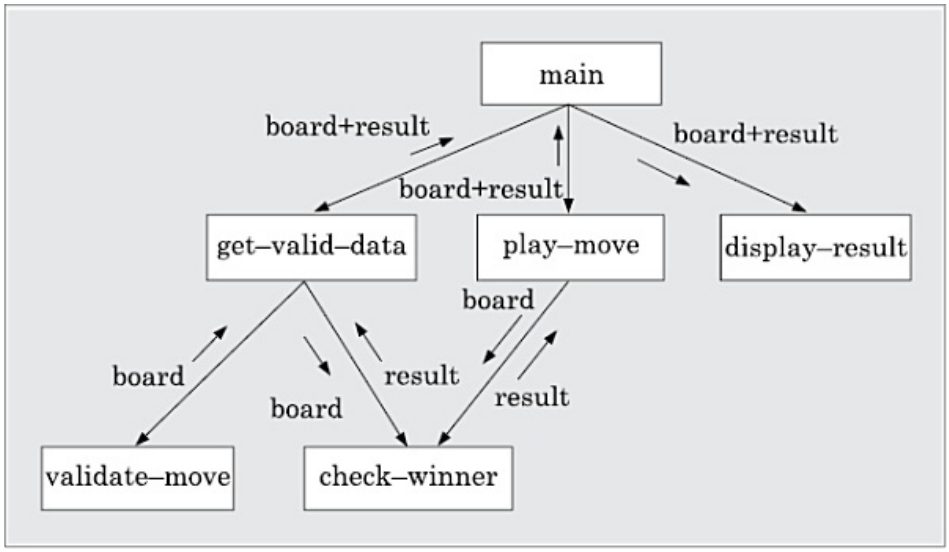

Figure 6.20: Structure chart for Example 6.7.

As in transform analysis, first all data entering into the DFD need to be identified. In a transaction-driven system, different data items may pass through different computation paths through the DFD. This is in contrast to a transform centered system where each data item entering the DFD goes through the same processing steps. Each different way in which input data is processed is a transaction. A simple way to identify a transaction is the following. Check the input data. The number of bubbles on which the input data to the DFD are incident defines the number of transactions. However, some transactions may not require any input data. These transactions can be identified based on the experience gained from solving a large number of examples.

For each identified transaction, trace the input data to the output. All the traversed bubbles belong to the transaction. These bubbles should be mapped to the same module on the structure chart. In the structure chart, draw a root module and below this module draw each identified transaction as a module. Every transaction carries a tag identifying its type. Transaction analysis uses this tag to divide the system into transaction modules and a transaction-center module.

Example 6.8 Draw the structure chart for the Supermarket Prize Scheme software of Example 6.3.

The structure chart for the Supermarket Prize Scheme software is shown in Figure 6.21.

Example 6.9 Draw the structure chart for the *trade-house automation system* (TAS) software of Example 6.4.

The structure chart for the *trade-house automation system* (TAS) software of

Example 6.4 is shown in Figure 6.22.

By observing the level 1 DFD of Figure 6.14, we can see that the data input to the diagram are handled by different bubbles and therefore transaction analysis is applicable to this DFD. Input data to this DFD are handled in three different ways (accept-order, accept- indent-request, and handle-query), we have three different transactions corresponding to these as shown in Figure 6.22.

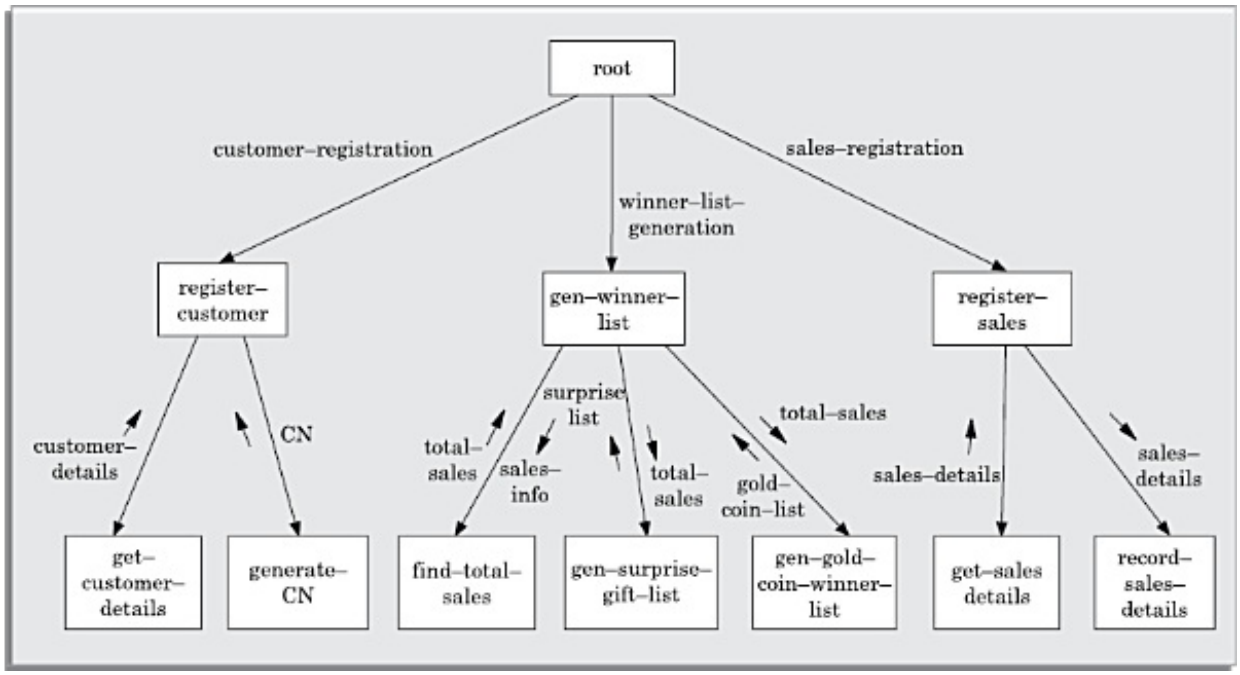

Figure 6.21: Structure chart for Example 6.8.

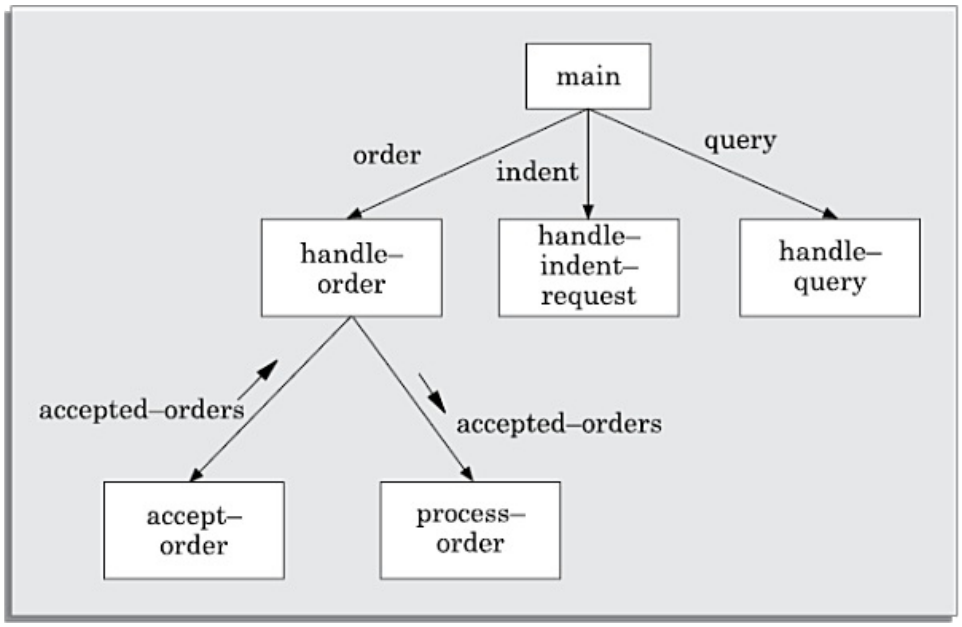

Figure 6.22: Structure chart for Example 6.9.

Word of caution

We should view transform and transaction analyses as guidelines, rather than rules. We should apply these guidelines in the context of the problem and handle the pathogenic cases carefully.

Example 6.10 Draw the structure chart for the personal library software of Example 6.6.

The structure chart for the personal library software is shown in Figure 6.23.

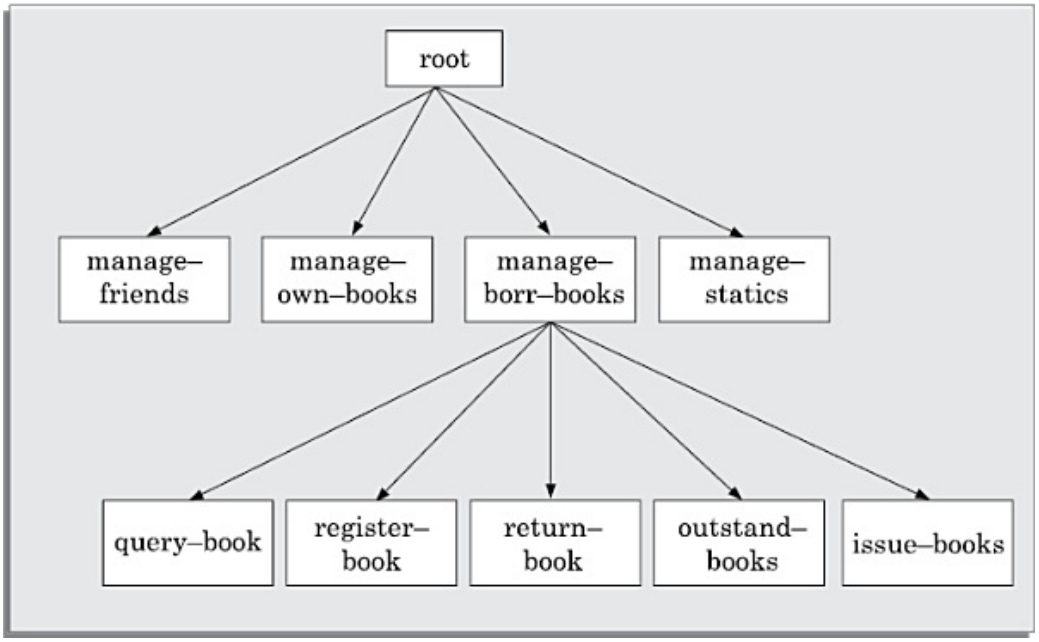

Figure 6.23: Structure chart for Example 6.10.

6.5 DETAILED DESIGN

During detailed design the pseudo code description of the processing and the different data structures are designed for the different modules of the structure chart. These are usually described in the form of module specifications (MSPEC). MSPEC is usually written using structured English. The MSPEC for the non-leaf modules describe the different conditions under which the responsibilities are delegated to the lower-level modules. The MSPEC for the leaf-level modules should describe in algorithmic form how the primitive processing steps are carried out. To develop the MSPEC of a module, it is usually necessary to refer to the DFD model and the SRS document to determine the functionality of the module.

6.6 DESIGN REVIEW

After a design is complete, the design is required to be reviewed. The review team usually consists of members with design, implementation, testing, and maintenance perspectives, who may or may not be the members of the development team. Normally, members of the team who would code the design, and test the code, the analysts, and the maintainers attend the review meeting. The review team checks the design documents especially for the following aspects:

Traceability: Whether each bubble of the DFD can be traced to some module in the structure chart and vice versa. They check whether each functional requirement in the SRS document can be traced to some bubble in the DFD model and vice versa.

Correctness: Whether all the algorithms and data structures of the detailed design are correct.

Maintainability: Whether the design can be easily maintained in future.

Implementation: Whether the design can be easily and efficiently be implemented.

After the points raised by the reviewers is addressed by the designers, the design document becomes ready for implementation.

SUMMARY

- In this chapter, we discussed a sample function-oriented software design methodology called structured analysis/structured design (SA/SD) which incorporates features of some important design methodologies.
- Methodologies like SA/SD give us a recipe for developing a good design according to the different goodness criteria we had discussed in Chapter 5. item SA/SD consists of two important parts—structured analysis and structured design.
- The goal of structured analysis is to perform a functional decomposition of the system. Results of structured analysis is represented using data flow diagrams (DFDs). The DFD representation is difficult to implement using a traditional programming language. The DFD representation can be systematically be transformed to structure chart representation. The structure chart representation can be easily implemented using a conventional programming language.
- During structured design, the DFD representation obtained during

structured analysis is transformed into a structure chart representation.

- Several CASE tools are available to support the software design process carried out using the important function-oriented design methodologies. In addition to laying out the DFDs, structure charts, maintaining the data dictionary, and helping in traceability analysis, these CASE tools can also perform some elementary consistency checking, e.g., they can usually check whether a DFD is balanced or not.

EXERCISES

1. Choose the correct option:
   (a) A data flow diagram represents:
   (i) The conditions based on which a data may be processed
   (ii) The order in which different activities are carried out
   (iii) The transformation of data through processing stations
   (iv) The order in which various functions of a program are invoked
   (b) A DFD depicts which of the following?
   (i) Flow of data
   (ii) Flow of control
   (iii) Flow of statements
   (iv) None of the above
   (c) Which of the following statements is not true of data flow diagrams (DFDs)?
   (i) Hierarchical diagram.
   (ii) Represent code structure
   (iii) Do not represent decisions and control flows.
   (iv) Represent functional decomposition.
   (d) In a procedural design approach, during the detailed design stage, which of the following is undertaken?
   (i) Module structure is designed
   (ii) Data flow representation is developed
   (iii) Data structures and algorithms for the individual modules are developed
   (iv) Structure chart is developed
2. What do you understand by the term "top-down decomposition" in the context of function-oriented design?
3. Distinguish between a data flow diagram (DFD) and a flow chart.
4. Differentiate between structured analysis and structured design in the

context of function-oriented design.

5. Point out the important differences between a structure chart and a flow chart as design representation techniques.
6. What do you mean by the term data dictionary in the context of structured analysis?
   How is a data dictionary useful during software development and maintenance?
7. Construct the DFD representation for the following program:

```
main(){                    .    f(int a,int b){    .    f1(){
   int a[100];             .      if (a>b) f1();   .       return;}
   for(i=0;i<100;i++)      .      else f2();       .
      f(a[i],a[i+1]);      .    }                  .    f2(){
}                          .                       .       return;}
```

8. Explain how a DFD model of software can be created from its source code.
9. What do you understand by the terms "structured analysis" and "structured design"?

What are the main objectives of "structured analysis" and "structured design"?

10. Explain how the DFD model can help one understand the working of a software system.
11. State whether the following statement is **TRUE** or **FALSE**. "The essence of any good function-oriented design principle is to map similar functions into a module." Give reasons behind your answers.
12. Identify the correct statement. Give reasoning behind your choice.
    (a) A DFD model essentially represents the data and control relationships among program elements.
    (b) A DFD model of a system usually comprises many DFDs.
    (c) The DFD model is the design model of a system.
    (d) A DFD model cannot represent a system's file data storage.
13. What do you mean by balancing a DFD? Illustrate your answer with a suitable example.
14. What are the main shortcomings of data flow diagram (DFD) as a tool for performing structured analysis?
15. Why is design reviews important? Suppose you are required to review a SA/SD document, make a list of items that can be used as a checklist for carrying out the review.
16. What do you understand by design review? What kinds of mistakes are normally pointed out by the reviewers?

17. (a) Draw a labelled DFD for the following **time management software.** Clearly show the context diagram and its hierarchical decompositions up to level 2. (Note: Context diagram is the Level 0 DFD).

A company needs to develop a time management system for its executives. The software should let the executives register their daily appointment schedules. The information to be stored includes person(s) with whom meeting is arranged, venue, the time and duration of the meeting, and the purpose (e.g., for a specific project work). When a meeting involving many executives needs to be organised, the system should automatically find a common slot in the diaries of the concerned executives, and arrange a meeting (i.e., make relevant entries in the diaries of all the concerned executives) at that time. It should also inform the concerned executives about the scheduled meeting through e-mail. If no common slot is available, TMS should help the secretary to rearrange the appointments of the executives in consultation with the concerned executives for making room for a common slot. To help the executives check their schedules for a particular day the system should have a very easy-to-use graphical interface. Since the executives and the secretaries have their own desktop computers, the time management software should be able to serve several remote requests simultaneously. Many of the executives are relative novices in computer usage. Everyday morning the time management software should e-mail every executive his appointments for the day. Besides registering their appointments and meetings, the executives might mark periods for which they plan to be on leave. Also, executives might plan out the important jobs they need to do on any day at different hours and post it in their daily list of engagements. Other features to be supported by the TMS are the following—TMS should be able to provide several types of statistics such as which executive spent how much time on meetings. For which project how many meetings were organised for what duration and how many man-hours were devoted to it. Also, it should be able to display for any given period of time the fraction of time that on the average each executive spent on meetings.

(b) Using the DFD you have developed for Part (a) of this question, develop the structured design for the time management software.

18. A hotel has a certain number of rooms. Each room can be either single bed or double bed type and may be AC or non-AC type. The rooms have different rates depending on whether they are of single or double, AC or

Non-AC types. The room tariff however may vary during different parts of the year depending up on the occupancy rate. For this, the computer should be able to display the average occupancy rate for a given month, so that the manager can revise the room tariff for the next month either upwards or downwards by a certain percentage. Perform structured analysis and structured design for this **Hotel Automation Software**—software that would automate the book keeping activities of a 5-star hotel.

Guests can reserve rooms in advance or can reserve rooms on the spot depending upon availability of rooms. The receptionist would enter data pertaining to guests such as their arrival time, advance paid, approximate duration of stay, and the type of the room required. Depending on this data and subject to the availability of a suitable room, the computer would allot a room number to the guest and assign a unique token number to each guest. If the guest cannot be accommodated, the computer generates an apology message. The hotel catering services manager would input the quantity and type of food items as and when consumed by the guest, the token number of the guest, and the corresponding date and time. When a customer prepares to check-out, the hotel automation software should generate the entire bill for the customer and also print the balance amount payable by him. During check-out, guests can opt to register themselves for a frequent guests programme. Frequent guests should be issued an identity number which helps them to get special discounts on their bills.

19. Perform structured analysis and structured design (SA/SD) for a software to be developed for automating various book keeping activities of a small book shop. From a discussion with the owner of the book shop, the following user requirements for this **Book-shop Automation Software (BAS)**—have been arrived at:

BAS should help the customers query whether a book is in stock. The users can query the availability of a book either by using the book title or by using the name of the author. If the book is not currently being sold by the book-shop, then the customer is asked to enter full details of the book for procurement of the book in future. The customer can also provide his e-mail address, so that he can be intimated automatically by the software as and when the book copies are received. If a book is in stock, the exact number of copies available and the rack number in which the book is located should be displayed. If a book is not in stock,

the query for the book is used to increment a request field for the book. The manager can periodically view the request field of the books to arrive at a rough estimate regarding the current demand for different books. BAS should maintain the price of various books. As soon as a customer selects his books for purchase, the sales clerk would enter the ISBN numbers of the books. BAS should update the stock, and generate the sales receipt for the book. BAS should allow employees to update the inventory whenever new supply arrives. Also upon request by the owner of the book shop, BAS should generate sales statistics (viz., book name, publisher, ISBN number, number of copies sold, and the sales revenue) for any period. The sales statistics will help the owner to know the exact business done over any period of time and also to determine inventory level required for various books. The inventory level required for a book is equal to the number of copies of the book sold over a period of one week multiplied by the average number of weeks it takes to procure the book from its publisher. Every day the book shop owner would give a command for the BAS to print the books which have fallen below the threshold and the number of copies to be procured along with the full address of the publisher.

20. Perform structured analysis and structured design for the following **City Corporation Automation Software (CCAS)** to be developed for automating various book keeping activities associated with various responsibilities of the Municipal Corporation of a large city.

    A city corporation wishes to develop a web-site using which the residents can get information on various facilities being provided by the corporate to the citizens. Since the city population exceeds 5 lakh, the maximum number of concurrent clicks can be upto 10 clicks per second. The corporation also plans to use the same web site for its road maintenance activity.

    A city corporation has branch offices at different suburbs of the city. Residents would raise repair requests for different roads of the city on line. The supervisor at each branch office should be able to view all new repair requests pertaining to his area. Soon after a repair request is raised, a supervisor visits the road and studies the severity of road condition. Depending on the severity of the road condition and the type of the locality (e.g., commercial area, busy area, relatively deserted area, etc.), he determines the priority for carrying out this repair work. The supervisor also estimates the raw material requirement for carrying

out the repair work, the types and number of machines required, and the number and types of personnel required. The supervisor enters this information through a special login in the web site. Based on this data, the system should schedule the repair of the road depending on the priority of the repair work and subject to the availability of raw material, machines, and personnel. This schedule report is used by the supervisor to direct different repair work. The manpower and machine that are available are entered by the city corporation administrator. He can change these data any time. Of course, any change to the available manpower and machine would require a reschedule of the projects. The progress of the work is entered periodically by the supervisor which can be seen by the citizens in the web site.

The mayor of the city can request for various road repair statistics such as the number and type of repairs carried out over a period of time and the repair work outstanding at any point of time and the utilisation statistics of the repair manpower and machine over any given period of time.

21. Perform structured analysis and structured design for developing the following **Restaurant Automation System** using the SA/SD technique.

A restaurant owner wants to computerise his order processing, billing, and accounting activities. H e also expects the computer to generate statistical report about sales of different items. A major goal of this computerisation is to make supply ordering more accurate so that the problem of excess inventory is avoided as well as the problem of non-availability of ingredients required to satisfy orders for some popular items is minimised. The computer should maintain the prices of all the items and also support changing the prices by the manager. Whenever any item is sold, the sales clerk would enter the item code and the quantity sold. The computer should generate bills whenever food items are sold. Whenever ingredients are issued for preparation of food items, the data is to be entered into the computer. Purchase orders are generated on a daily basis, whenever the stock for any ingredient falls below a threshold value. The computer should calculate the threshold value for each item based on the average consumption of this ingredient for the past three days and assuming that a minimum of two days stock must be maintained for all ingredients. Whenever the ordered ingredients arrive, the invoice data regarding the quantity and price is entered. If sufficient cash balance is available, the computer should print

cheques immediately against invoice. Monthly sales receipt and expenses data should be generated whenever the manger would request to see them.

22. Perform structured analysis and design for the following **Judiciary Information System (JIS) software.**

The attorney general's office has requested us to develop a Judiciary Information System (JIS), to help handle court cases and also to make the past court cases easily accessible to the lawyers and judges. For each court case, the name of the defendant, defendant's address, the crime type (e.g., theft, arson, etc.), when committed (date), where committed (location), name of the arresting officer, and the date of the arrest are entered by the court registrar. Each court case is identified by a unique case identification number (CIN) which is generated by the computer. The registrar assigns a date of hearing for each case. For this the registrar expects the computer to display the vacant slots on any working day during which the case can be scheduled. Each time a case is adjourned, the reason for adjournment is entered by the registrar and he assigns a new hearing date. If hearing takes place on any day for a case, the registrar enters the summary of the court proceedings and assigns a new hearing date. Also, on completion of a court case, the summary of the judgment is recorded and the case is closed but the details of the case is maintained for future reference. Other data maintained about a case include the name of the presiding judge, the public prosecutor, the starting date, and the expected completion date of a trial. The judges should be able to browse through the old cases for guidance on their judgment. The lawyers should also be permitted to browse old cases, but should be charged for each old case they browse. Using the JIS software, the Registrar of the court should be able to query the following:

(a) The currently pending court cases. In response to this query, the computer should print out the pending cases sorted by CIN. For each pending case, the following data should be listed—the date in which the case started, the defendant's name, address, crime details, the lawyer's name, the public prosecutor's name, and the attending judge's name.

(b) The cases that have been resolved over any given period. The output in this case should chronologically list the starting date of the case, the CIN, the date on which the judgment was delivered, the

name of the attending judge, and the judgment summary.
(c) The cases that are coming up for hearing on a particular date. (d) The status of any particular case (cases are identified by CIN).

23. The different activities of the library of our institute pertaining to the issue and return of the books by the members of the library and various queries regarding books as listed below are to be automated. Perform structured analysis and structured design for this Library Information System (LIS) software:

- The library has 10,000 books. Each book is assigned a unique identification number (called ISBN number). The Library clerk should be able to enter the details of the book into the LIS through a suitable interface.
- There are four categories of members of the library—undergraduate students, post graduate students, research scholars, and faculty members.
- Each library member is assigned a unique library membership code number.
- Each undergraduate student can issue up to 2 books for 1 month duration.
- Each postgraduate student can issue up to 4 books for 1 month duration.
- Each research scholar can issue up to 6 books for 3 months duration.
- Each faculty member can issue up to 10 books for six months duration.
- The LIS should answer user queries regarding whether a particular book is available. If a book is available, LIS should display the rack number in which the book is available and the number of copies available.
- LIS registers each book issued to a member. When a member returns a book, LIS deletes the book from the member's account and makes the book available for future issue.
- Members should be allowed to reserve books which have been issued. When such a reserved book is returned, LIS should print a slip for the concerned member to get the book issued and should disallow issue of the book to any other

member for a period of seven days or until the member who has reserved the books gets it issued.

- When a member returns a book, LIS prints a bill for the penalty charge for overdue books. LIS calculates the penalty charge by multiplying the number of days the book is overdue by the penalty rate.
- LIS prints reminder messages for the members against whom books are over due, upon a request by the Librarian.
- LIS should allow the Librarian to create and delete member records. Each member should be allocated a unique membership identification number which the member can use to issue, return, and reserve books.

24. Perform the SA/SD for the following word processing software.

- The word processing software should be able to read text from an ASCII file or HTML file and store the formatted text as HTML files in the disk.
- The word processing software should ask the user about the number of characters in an output line of the formatted text. The user should be allowed to select any number between 1 and 132.
- The word processing software should process the input text in the following way.

– Each output line is to contain exactly the number of characters specified by the user (including blanks).
– The word processing software is to both left and right justify the text so that there are no blanks at the left- and right-hand ends of lines except the first and possibly the last lines of paragraphs. The word processing software should do this by inserting extra blanks between words.
– The input text from the ASCII file should consist of words separated by one or more blanks and a special character PP, which denotes the end of a paragraph and the beginning of another.
– The first line of each paragraph should be indented by five spaces and should be right justified.
– The last line of each paragraph should be left justified.

- The user should be able to browse through the document and add, modify or delete words. He/she should also be able to mark any word as bold, italic, superscript, or subscript.
- The user can request to see the number of characters, words, lines, and paragraphs used in the document.
- The user should be able to save his documents under a name specified by him.

25. It is required to develop a graphics editor software package using which one can create/modify several types of graphics entities. In summary, the graphics editor should support the following features: (Those who are not familiar with any graphics editor, please look at the Graphics Drawing features available in either MS-Word or PowerPoint software. You can also examine any other Graphical Drawing package accessible to you. An understanding of the standard features of a Graphics Editor will help you understand the different features required.)

    - The graphics editor should support creating several types of geometric objects such as circles, ellipses, rectangles, lines, text, and polygons.
    - Any created object can b e selecte d by clicking a mouse button on the object. A selected object should be shown in a highlighted color.
    - A selected object can be edited, i.e., its associated characteristics such as its geometric shape, location, color, fill style, line width, line style, etc. can be changed. For texts, the text content can be changed.
    - A selected object can be copied, moved, or deleted.
    - The graphics editor should allow the user to save his created drawings on the disk under a name he would specify. The graphics editor should also support loading previously created drawings from the disk.
    - The user should be able to define any rectangular area on the screen to be zoomed to fill the entire screen.
    - A fit screen function makes the entire drawing fit the screen by automatically\ adjusting the zoom and pan values.
    - A pan function should allow the displayed drawing to be panned along any direction by a specified amount.

- The graphics editor should support grouping. A group is simply a set of drawing objects including other groups which when grouped behave as a single entity. This feature is especially useful when you wish to manipulate several entities in the same way. A drawing object can be a direct member of at most one group. It should be possible to perform several editing operations on a group such as move, delete, and copy.
- A set of 10 clip boards should be provided to which one can copy various types of selected entities (including groups) for future use in pasting these at different places when required.

26. Perform SA/SD for the following **Software component cataloguing software.**

Software component cataloguing software: The software component cataloguing software consists of a software components catalogue and various functions defined on this components catalogue. The software components catalogue should hold details of the components which are potentially reusable. The reusable components can be either design or code. The design might have been constructed using different design notations such as UML,ERD,structured design, etc. Similarly, the code might have been written using different programming languages. A cataloguer may enter components in the catalogue, may delete components from the catalogue, and may associate reuse information with a catalogue component in the form of a set of key words. A user of the catalogue may query about the availability of a component using certain key words to describe the component. In order to help manage the component catalogue (i.e., periodically purge the unused components) the cataloguing software should maintain information such as how many times a component has been used, and how many times the component has come up in a query but not used. Since the number of components usually tend to be very high, it is desirable to be able to classify the different types of components hierarchically. A user should be able to browse the components in each category.

27. The manager of a supermarket wants us to develop an automation software. The supermarket stocks a set of items. Customers pick up their desired items from the different counters in required quantities. The customers present these items to the sales clerk. The sales clerk enters

the code number of these items along with the respective quantity/units. Perform structured analysis and structured design for developing this **Supermarket Automation Software (SAS).**

- SAS should at the end of a sales transaction print the bill containing the serial number of the sales transaction, the name of the item, code number, quantity, unit price, and item price. The bill should indicate the total amount payable.
- SAS should maintain the inventory of the various items of the supermarket. The manager upon query should be able to see the inventory details. In order to support inventory management, the inventory of an item should be decreased whenever an item is sold. SAS should also support an option by which an employee can update the inventory when new supply arrives.
- SAS should support printing the sales statistics for every item the supermarket deals with for any particular day or any particular period. The sales statistics should indicate the quantity of an item sold, the price realised, and the profit.
- The manager of the supermarket should be able to change the price at which an item is sold as the prices of the different items vary on a day-to-day basis.

28. A transport company wishes to computerise various book keeping activities associated with its operations. Perform structured analysis and structured design for developing the **Transport Company Computerisation (TCC) software**:

- A transport company owns a number of trucks.
- The transport company has its head office located at the capital and has branch offices at several other cities.
- The transport company receives consignments of various sizes at (measured in cubic meters) its different offices to be forwarded to different branch offices across the country.
- Once the consignment arrives at the office of the transport company, the details of the volume, destination address, sender address, etc., are entered into the computer. The computer would compute the transport charge depending

upon the volume of the consignment and its destination and would issue a bill for the consignment.

- Once the volume of any particular destination becomes 500 cubic meters, the Computerisation system should automatically allot the next available truck.
- A truck stays with the branch office until the branch office has enough cargo to load the truck fully.
- The manager should be able to view the status of different trucks at any time.
- The manager should be able to view truck usage over a given period of time.
- When a truck is available and the required consignment is available for dispatch, the computer system should print the details of the consignment number, volume, sender's name and address, and the receiver's name and address to be forwarded along with the truck.
- The manager of the transport company can query the status of any particular consignment and the details of volume of consignments handled to any particular destination and the corresponding revenue generated.
- The manager should also be able to view the average waiting period for different consignments. This statistics is important for him since he normally orders new trucks when the average waiting period for consignments becomes high due to non- availability of trucks. Also, the manager would like to see the average idle time of the truck in the branch for a given period for future planning.

29. Draw level 0 (context level) and level 1 data flow diagram for the following students' academic record management software.

- A set of courses are created. Each course consists of a unique course number, number of credits, and the syllabus.
- Students are admitted to courses. Each students' details include his roll number, address, semester number and the courses registered for the semester.
- The marks of student for various units he credited are keyed in.

- Once the marks are keyed in, the semester weighted average (SWA) is calculated.
- The recent marks of the student are added to his previous marks and a weighted average based on the credit points for various units is calculated.
- The marks for the current semester are formatted and printed.
- The SWA appears on the report.
- A check must be made to determine if a student should be placed on the Vice- Chancellor's list. This is determined based on whether a student scores an SWA of 85 or higher.
- If the SWA is lower than 50, the student is placed on a conditional standing.

30. Perform structured analysis and structured design (SA/SD) for the following **CASE tool for Structured Analysis Software** to be developed for automating various activities associated with developing a CASE tool for structured software analysis.

    - The case tool should support a graphical interface and the following features.
    - The user should be able to draw bubbles, data stores, and entities and connect them using data flow arrows. The data flow arrows are annotated by the corresponding data names.
    - Should support editing the data flow diagram.
    - Should be able to create the diagram hierarchically.
    - The user should be able to determine balancing errors whenever required.
    - The software should be able to create the data dictionary automatically.
    - Should support printing the diagram on a variety of printers.
    - Should support querying the data items and function names. The diagrams matching the query should be shown.

31. Perform structured analysis and structured design (SA/SD) for a software to be developed for automating various activities associated with developing a CASE tool for structured software design. The summary of the requirements for this **CASE tool for Structured Design** are

the following:

- The case tool should support a graphical interface and the following features.
- It should be possible to import the DFD model developed by another program. The user should be able to apply the transform and transaction analysis to the imported DFD model.
- The user should be able to draw modules, control arrows, and data flow arrows. Also symbol for library modules should be provided. The data flow arrows are annotated with the corresponding data name.
- The modules should be organised in hierarchical levels.
- The user should be able to modify his design. Please note that when he deletes a data flow arrow, its annotated data name automatically gets deleted.
- For large software, modules may be hierarchically organised and clicking on a module should be able to show its internal organisation.
- The user should be able to save his design and also be able to load previously created designs.

32. The local newspaper and magazine delivery agency has asked us to develop a software for him to automate various clerical activities associated with his business. Perform the structured analysis and design for this **Newspaper Agency Automation Software.**

- This software is to be used by the manager of the news agency and his delivery persons.
- For each delivery person, the system must print each day the publications to be delivered to each address.
- The customers usually subscribe one or more news papers and magazines. They are allowed to change their subscription notice by giving one week's advance notice. Customers should be able to initiate new subscriptions and suspend subscription for a particular item either temporarily or permanently through a web browser. Considering the large customer base, at least 10 concurrent customer

accesses should be supported.

- For each delivery person, the system must print each day the publications to be delivered to each address.
- The system should also print for the news agent the information regarding who received what and a summary information of the current month.
- At the beginning of every month bills are printed by the system to be delivered to the customers. These bills should be computed by the system automatically.
- The customers may ask for stopping the deliveries to them for certain periods when they go out of station.
- Customers may request to subscribe new newspapers/magazines, modify their subscription list, or stop their subscription altogether.
- Customers usually pay their monthly dues either by cheque or cash. Once the cheque number or cash received is entered in the system, receipt for the customer should be printed.
- If any customer has any outstanding due for one month, a polite reminder message is printed for him and his subscription is discontinued if his dues remain outstanding for periods of more than two months.
- The software should compute and print out the amount payable to each delivery boy. Each delivery boy gets 2.5 per cent of the value of the publications delivered by him.

33. Perform SA/SD for the following **University Department Information System.** This software concerns automating the activities of department offices of universities. Department offices in different universities do a lot of book-keeping activities the software to be developed targets to automate these activities.

- Various details regarding each student such as his name, address, course registered, etc. are entered at the time he takes admission.
- At the beginning of every semester, students do course registration. The information system should allow the department secretary to enter data regarding student course registrations. As the secretary enters the roll number of each

student, the computer system should bring up a form for the corresponding student and should keep track of courses he has already completed and the courses he has back-log, etc.

- At the end of the semester, the instructors leave their grading information at the office which the secretary enter in the computer. The information system should be able to compute the grade point average for each student for the semester and his cumulative grade point average (CGPA) and print the grade sheet for each student.
- The information system also keep s track of a inventories of the Department, such as equipments, their location, furnitures, etc.
- The Department has an yearly grant and the Department spends it in buying equipments, books, stationery items, etc. Also, in addition to the annual grant that the Department gets from the University, it gets funds from different consultancy service it provides to different organisations. It is necessary that the Department information system keeps track of the Department accounts.
- The information system should also keep track of the research projects of the Department, publications by the faculties, etc. These information are keyed in by the secretary of the Department.
- The information system should support querying the up-to-date details about every student by inputting his roll number. It should also support querying the details of the cash book account. The output of this query should include the income, expenditure, and balance.

34. Perform SA/SD to develop a software to automate the activities of a small automobile spare parts shop. The small automobile spare parts shop sells the spare parts for a vehicles of several makes and models. Also, each spare part is typically manufactured by several small industries. To stream line the sales and supply ordering, the shop owner has asked us to develop the following motor part shop software. Perform the SA/SD for this **Motor Part Shop Software (MPSS).**

The motor parts shop deals with large number of motor parts of various manufacturers and various vehicle types. Some of the motor parts are

very small and some are of reasonably large size. The shop owner maintains different parts in wall mounted and numbered racks.

The shop owner maintains as few inventory for each item as reasonable, to reduce inventory overheads after being inspired by the just-in-time (JIT) philosophy.

Thus, one important problem the shop owner faces is to be able to order items as soon as the number of items in the inventory reduces below a threshold value. The shop owner wants to maintain parts to be able to sustain selling for about one week. To calculate the threshold value for each item, the software must be able to calculate the average number of parts sales for one week for each part.

At the end of each day, the shop owner would request the computer to generate the items to be ordered. The computer should print out the part number, the amount required and the address of the vendor supplying the part.

The computer should also generate the revenue for each day and at the end of the month, the computer should generate a graph showing the sales for each day of the month.

35. Perform structured analysis and structured design for the following

Medicine Shop

- **Automation (MSA)** software:
- A retail medicine shop deals with a large number of medicines procured from various manufacturers. The shop owner maintains different medicines in wall mounted and numbered racks.
- The shop owner maintains as few inventory for each item as reasonable, to reduce inventory overheads after being inspired by the just-in-time (JIT) philosophy.
- Thus, one important problem the shop owner faces is to be able to order items as soon as the number of items in the inventory reduces below a threshold value. The shop owner wants to maintain medicines to be able to sustain selling for about one week. To calculate the threshold value for each item, the software must be able to calculate the average number of medicines sales for one week for each part.
- At the end of each day, the shop owner would request the computer to generate the items to be ordered. The

computer should print out the medicine description, the quantity required, and the address of the vendor supplying the medicine. The shop owner should be able to store the name, address, and the code numbers of the medicines that each vendor deals with.

- Whenever new supply arrives, the shop owner would enter the item code number, quantity, batch number, expiry date, and the vendor number. The software should print out a cheque favouring the vendor for the items supplied.
- When the shop owner procures new medicines it had not dealt with earlier, he should be able to enter the details of the medicine such as the medicine trade name, generic name, vendors who can supply this medicine, unit selling and purchasing price. The computer should generate a code number for this medicine which the shop owner would paste the code number in the rack where this medicine would be stored. The shop owner should be able to query about a medicine either using its generic name or the trade name and the software should display its code number and the quantity present.
- At the end of every day the shop owner would give a command to generate the list of medicines which have expired. It should also prepare a vendor-wise list of the expired items so that the shop owner can ask the vendor to replace these items. Currently, this activity alone takes a tremendous amount of labour on the part of the shop owner and is a major motivatour for the automation endeavour.
- Whenever any sales occurs, the shop owner would enter the code number of each medicine and the corresponding quantity sold. The MSA should print out the cash receipt.
- The computer should also generate the revenue and profit for any given period. It should also show vendor-wise payments for the period.

36. The IIT students' Hall Management Center (HMC) has requested us to develop the following software to automate various book-keeping activities associated with its day- to-day operations.

- After a student takes admission, he/she presents a note from the admission unit, along with his/her name, permanent address, contact telephone number, and a photograph. He/she is then allotted a hall, and also a specific room number. A letter indicating this allotted room is issued to the concerned student.
- Students incur mess charges every month. The mess manager would input to the software the total charges for each student in a month on mess account.
- Each room has a fixed room rent. The newly constructed halls have higher rent compared to some of the older halls. Twin sharing rooms have lower rent.
- Each hall provides certain amenities to the students such as reading rooms, play rooms, TV room, etc. A fixed amount is levied on each student on this count.
- The total amount collected from each student of a hall towards mess charges is handed over to the mess manager every month. For this, the computer needs to print a sheet with the total amount due to each mess manager is printed. Printed cheques are issued to each manager and signatures are obtained from them on the sheet.
- Whenever a student comes to pay his dues, his total due is computed as the sum of mess charge, amenity charge, and room rent.
- The students should be able to raise various types of complaints using a web browser in their room or in the Lab. The complaints can be repair requests such as fused lights, non-functional water taps, non-functional water filters, room repair, etc. They can also register complaints regarding the behaviour of attendants, mess staff, etc. For this round-the-clock operation of the software is required.
- The HMC receives an annual grant from the Institute for staff salary and the upkeep of the halls and gardens. The HMC chairman should be provided support for distribution of the grant among the different halls. The Wardens of different halls should be able to enter their expenditure details against the allocations.
- The controlling warden should be able to view the overall

room occupancy.

- The warden of each hall should be able to find out the occupancy of his hall. He should also be able to view the complaints raised by the students and post his action taken report (ATR) to each complaint.
- The halls employ attendants and gardeners. These temporary employees receive a fixed pay on a per day basis. The Hall clerk enters any leave taken by an attendant or a gardener from at the terminal located at the hall office. At the end of every month a consolidated list of salary payable to each employee of the hall along with cheques for each employee is printed out.
- The HMC incurs petty expenses such as repair works carried out, newspaper and magazine subscriptions, etc. It should be possible to enter these expenses.
- Whenever a new staff is recruited his details including his daily pay is entered. Whenever a staff leaves, it should be possible to delete his records.
- The warden should be able to view the statement of accounts any time. The warden would take a print out of the annual consolidated statement of accounts, sign and submit it to the Institute administration for approval and audit verification.

37. **IIT security software:** The security office of IIT is in need of a software to control and monitor the vehicular traffic into and out of the campus. The functionalities required of the software are as follows—Each vehicle in the IIT campus would be registered with the system. For this, each of the faculty, staff, and students owning one or more vehicles would have to fill up a form at the security office detailing the vehicle registration numbers, models, and other relevant details for the vehicles that they own. These data would be entered into the computer by a security staff after a due diligence check.

Each time a vehicle enters or leaves the campus, a camera mounted near the check gate would determine the registration number of an incoming (or outgoing) vehicle and the model of the vehicle and input into the system. If the vehicle is a campus vehicle, then the check gate should lift automatically to let it in or out, as the case may be. Various

details regarding the entry and exit of a campus vehicle such as its number, owner, date and time of entry/exit would be stored in the database for statistical purposes. For each outside vehicle entering the campus, the driver would be required to fill-up a form detailing the purpose of entry. This information would immediately be entered by the security personnel at the gate and along with this information, the information obtained from the camera such as the vehicle's model number, registration number, and photograph would be stored in the database. When an outside vehicle leaves the campus, the exit details such as date and time of exit would automatically be registered in the database. For any external vehicle that remains inside the campus for more than 8 hours, the driver would be stopped by the security personnel manning the gate, queried to satisfaction, and the response would be entered into the system.

Considering that there have been several incidents of speeding and rough driving in the past, the security personnel manning various traffic intersections and other sensitive points of the campus would be empowered to telephone the registration number of an errant vehicle to the main gate. The security personnel at the main gate would enter this information into the computer. The driver of an errant vehicle would be quizzed at the check gate during exit and the vehicle's future entry would be barred if the response is not found to be satisfactory. For each errant campus vehicle, the driver would be quizzed during the next exit, and if the response is not found to be satisfactory, a letter should get issued to the dean (campus affairs) (detailing the date, time, traffic point at which the incident occurred, and the nature of the offence) to deal with the concerned staff or student, as the case might be.

The security officer should be able to view the statistics pertaining to the total number of vehicles going in and coming out of the campus (over a day, month or year) and the total number of vehicles currently inside the campus has left campus, and if it indeed has, its time of departure should be displayed. The security officer can query whether a particular vehicle is currently inside the campus. The security officer can also query the total number of vehicles owned by the residents of the campus. Since campus security is a critical operation, adequate safety against cyber attacks on the security software should be ensured. Also, considering the criticality of the operation, down times of more than 5 minutes would normally not be unacceptable.

Implementation simplification: While writing code, the commands for the check gate need only be displayed on the terminal and the inputs from the camera can be simulated through keyboard entry.

38. **Courier company computerisation (CCC) software:** A courier company wishes to computerise various book keeping activities associated with its daily operation. The courier company has branches in most important towns in India. It is proposed that the different branch offices be equipped with a computer and printer each. The developed software would be deployed on the computer at each branch office and linked though the Internet. The other details are as follows:

    - At each of its branch office and other retail outlets, the courier company receives consignments of various weights and sizes (measured in cubic meters). The charges are at present Rs. 5,000 per cubic meter for distances upto 500 km. For larger distances, 10 per cent additional charge is levied for every 100 km or part there of. For packets weighing more than 100 kg per cubic meter, an additional 10 per cent levy is charged for every 20 kg/cubic meter. For small articles and letters, Rs. 50 per 100 gms is charged. At present, only those packets having destination to a city where a branch office is located is accepted.
    - When a customer tries to book a consignment at any of the retail points or branch offices, the details of the consignment such as its volume, weight, destination address, sender address, etc. are entered into the computer by the sales clerk. The computer would compute the charges for the consignment and print a bill indicating a unique id indicating the consignment number, which is assigned to the consignment. A customer should be able to track the delivery status of the consignment on-line by using the unique id.
    - The courier company owns a number of trucks, which are used for transporting consignments between branch offices.
    - When the volume of consignments for any particular destination (branch office) becomes 500 cubic meters, the system should automatically allot the next available truck that is present at the branch office. Since, no consignment should get unduly delayed, whenever a consignment cannot

be dispatched to its destination within 3 days of its receipt, it should automatically be forwarded to any branch office that is closer to the destination. When a truck is allocated for dispatch of consignments to a branch office, the computer system should print the details of the consignment number, volume, sender's name and address, and the receiver's name and address. This print out is to be carried by the truck driver for monitoring and excise clearance purposes.

- When a truck reaches a branch office, its arrival status is updated. A truck stays with the branch office until the branch office has enough cargo to load the truck to at least 80 per cent of its capacity. The transport office at the branch office can enter the fuel and repair charges for a truck.
- The regular expenses of the courier company includes staff salaries, rental charges for the branch offices, and truck maintenance charges. The company judiciously uses its profits to set up new branch offices and to buy additional trucks.
- The software should maintain the details of each employee such as his name, address, telephone number, basic pay, and other allowances. It should help the account manager at each branch office to generate the pay slip of all the employees every month and automatically credit the salaries to their respective bank accounts.
- All payments and receipts are entered into the system. A consolidated profit-loss account (taking all the branches and the entire operation into account) is expected to be prepared by the system. The manager should be able to view branch-wise revenue generated, consignments handled, expenses, etc.
- The manager should be able to view the status of different trucks at any time, e.g. the branch office at which it is waiting, or the two branch offices between which it is currently transporting consignments.
- The manager should be able to view truck usage (overall as well as for individual trucks) over a given period of time. The truck usage is to be given in terms of load factor (average capacity utilisation) and number of kilometers covered over

the given period.

- The manager of the courier company can query the status of any particular consignment and the details o f volume of consignments transported between any two branches and the corresponding revenue generated.
- The manager should be able to view the average waiting period for the consignments over a given period of time (day, month, or year) and that for various source destination pairs. This statistics is important for the manager, since he normally orders new trucks when the average waiting period for consignments becomes high due to non-availability of trucks. Also, the manager would like to see the average idle times of trucks over a given period time for future planning.
- The courier company would like the software to be modular and highly configurable, so that it can sell copies of the software to other courier companies in the country.

39. **Students' auditorium management software:** A college has a large (800 seating capacity) auditorium. The college has entrusted the management of the auditorium to the students' society. The students' society needs the following software to efficiently manage the various shows conducted in the auditorium and to keep track of the accounts. The functionaries identified by the students' society to be responsible for the day-to-day operation of the software are the auditorium secretary and the president of the society.

Various types of social and cultural events are conducted in the auditorium. The auditorium secretary should have the overall authority of scheduling the shows, selecting and authorising the show managers, as well as the sales agents.

There are two categories of seats—balcony seats and ordinary seats. Normally balcony seats are more expensive in any show. The show manager fixes the prices of these two categories of seats for a specific show, depending on the popularity of a show. The show manager also determines the number of balcony and ordinary seats that can be put on sale. For each show, some seats are offered as complimentary gifts to important functionaries of the students' society and to VIPs which need to be entered into the system. The show manager also enters the show dates, the number of shows on any particular date and the show timings.

It is expected that the software would support a functionality to let the show manager configure the different show parameters.

The auditorium secretary appoints a set of sales agents. The sales agents get a commission of 1 per cent of the total sales that they make for any show. The system should let the spectators query the availability of different classes of seats for a show on-line. For the convenience of the spectators, two ways of seat booking are supported. If a spectator pays Rs. 1000, a unique 10 digit id is generated and given to him. He can use this id to book seats for the shows on-line by using a web browser. For each seat booked for a show using the unique id, he would get a 10 per cent discount. A spectator can also book a seat for a single show only through regular payment. For on-line booking, a spectator would indicate for the type of the seat required by him, the requested seat should be booked if available, and the software should support printing out the ticket showing the seat numbers allocated. A spectator should be able to cancel his booking before 3 clear days of the show. In this case, the ticket price is refunded to him after deducting Rs. 5 as the booking charge per ticket. If a ticket is returned at least before 1 day of a show, a booking charge of Rs. 10 is deducted for ordinary tickets and Rs. 15 is deducted for balcony tickets. When a cancellation is made on the day of the show, there is a 50 per cent deduction.

The show manager can at any time query about the percentage of seats booked for various classes of seats and the amount collected for each class. When a sales person makes a sale, the computer should record the sales person's id in the sales transaction. This information would help in computing the total amount collected by each sales person and the commission payable to each sales person. These data can be queried by the show manager. Also, any one should be able to view the various shows that are planned for the next one month and the rates of various categories of seats for a show by using a web browser. The show manager should be able to view the total amount collected for his show as well as the sales agent-wise collection figures.

The accounts clerk should be able to enter the various types of expenditures incurred for a show including payment to artists and auditorium maintenance charges. The computer should prepare a balance sheet for every show and a comprehensive up-to-date balance sheet for every year. The different types of balance sheets should be accessible to the president of the student society only. Since the

software product should be as much low cost as possible, it is proposed that the software should run on a high-end PC and built using free system software such as Linux and Apache web server.

40. **Travel agency management software:** A travel agency requires to automate various book-keeping activities associated with its operations. The agency owns a fleet of vehicles and it rents these out to customers. Currently the company has the following fleet of vehicles:

    - Ambassadors : 10 non-AC
    - Tata Indica : 30 AC
    - Tata Sumo : 5 AC
    - Maruti Omni : 10 non-AC
    - Maruti Esteem : 10 AC
    - Mahindra Xylo : 10 AC

    Only regular customers would be allowed to avail the on-line booking facility. To become a regular customer, a customer would need to deposit Rs. 5000 with the travel agency and also provide his address, phone number and few other details, which will be entered into the computer.

    When a regular customer makes an on-line request for hiring a vehicle, he would be prompted to enter the date for which travel is required, the destination, and duration for which the vehicle is required. He should then be displayed the types of vehicles that are available, and the charges. For every type of vehicle, there is a per hour charge, and a per kilometer charge. These information would also be displayed. A car can be rented for a minimum of 4 hours. Once a customer chooses a vehicle, the number of the allotted vehicle and the driver's mobile numbers would be displayed.

    After completion of travel, the customer would sign off the duty slip containing necessary information such as duration of hire and the kilometers covered and hand it over to the driver. This information would be entered into the computer system by the driver within 24 hours of completion of travel. The customer should be able to view the billing information as soon as the information in the duty slip have been entered into the computer. The amount chargeable to a customer is the maximum of per hour charge for the car times the number of hours used, plus the per kilometer charge times the number of kilometers run,

subject to a minimum amount decided by the charge for 4 hours use of the car. An AC vehicle of a particular category is charged 25 per cent more than a non-AC vehicle of the same category. There is a charge of Rs. 500 for every night halt regardless of the type of the vehicle.

The travel agency acquires new vehicles at regular intervals. It is necessary to support a functionality using which the manager would be able to add a vehicle to the fleet of the vehicles as and when a new vehicle is procured. Old cars are condemned and sold off.

It should be possible to delete the car from the fleet and add the sales proceeds to the account. A car which is currently with the company can be in one of three states—it may have gone for repair, it may be available, it may be rented out.

The manager should be able to view the following types of statistics—the price of the car, average amount of money spent on repairs for the car, average demand, revenue earned by renting out the car, and fuel consumption of the car. Based on these statistics, the company may take a decision about which vehicles are more profitable. The statistics can also be used to decide the rental charges for different types of vehicles.

41. **Students hall management center:** The IIT students' Hall Management Center (HMC) has requested us to develop the following software to automate various book- keeping activities associated with its day to day operations.

- After a student takes admission, he/she would present a note from the admission unit to the clerk at HMC, along with his/her name, permanent address, contact telephone number, and a photograph. He/she is then allotted a hall, and also a specific room number. A letter indicating this allotted room should be issued to the concerned student.
- Students incur mess charges every month. The mess manager would input to the software the total charges for each student in a month on mess account.
- Each room has a fixed room rent. The rooms are either single-seated or twin- sharing. The newly constructed halls have higher rent compared to some of the older halls. Twin sharing rooms have lower rent.
- Each hall provides certain amenities to the students such as reading rooms, play rooms, TV room, etc. A fixed amount is

levied on each student on this count.

- The total amount collected from each student of a hall towards mess charges is handed over to the mess manager every month. For this, the computer needs to print a sheet indicating the total amount due to each mess manager. Printed cheques are issued to each manager and signatures are obtained from each on the sheet.
- Whenever a student comes to pay his dues, his total due should be computed as the sum of mess charge, amenity charge, and room rent and displayed. The amount would be paid by the student either in cash or cheque, and this would be entered by the accounts clerk into the system.
- The students should be able to raise various types of complaints using a web browser in their room or in the Lab. The complaints can be repair requests such as fused lights, non-functional water taps, non-functional water filters, specific room repair, etc. They can also register complaints regarding the behaviour of attendants, mess staff, etc. For this, round-the-clock operation of the software is required and the down-time should be negligible. Considering that about 10,000 students live in hostels, the response time of the web site should be acceptable even under 1000 simultaneous clicks.
- The HMC receives an annual grant from the Institute for upkeep of the halls, gardens, and providing amenities to the students. The HMC chairman should be provided with a functionality that would support distribution of the grant among the different halls and cheques should be printed based on the grants made to the halls. The wardens of different halls should be able to enter their expenditure details against the allocations.
- The controlling warden should be able to view the overall room occupancy.
- The warden of each hall should be able to find out the total occupancy of his hall and the number of vacant seats. He should also be able to view the complaints raised by the students and post his action taken report (ATR) to each complaint.

- The halls employ attendants and gardeners. These temporary employees receive a fixed pay on a per day basis. The Hall clerk enters any leave taken by an attendant or a gardener from at the terminal located at the hall office. At the end of every month a consolidated list of salary payable to each employee of the hall along with cheques for each employee is printed out.
- The HMC incurs petty expenses such as repair works carried out, news paper and magazine subscriptions, etc. It should be possible to enter these expenses should be debited from its yearly grants.
- Whenever a new staff is recruited his details including his daily pay is entered. Whenever a staff leaves, it should be possible to delete his records. Upon a specific command from the controlling warden, the salaries for various employees of the HMC and halls should be computed and the salary slips and cheques should be printed for distribution to the employees.
- The warden should be able to view the statement of accounts any time. The warden would take a print out of the annual consolidated statement of accounts, sign and submit it to the Institute administration for approval and audit verification.

The software should be very secure to prevent the possibility of various types of frauds and financial irregularities.

42. Develop SA/SD diagrams for the following software required by a **video rental store**.

- The store has a large collection of video CDs and DVDs in VHS and MP4 format as well as music CDs.
- A person can become member by depositing Rs. 1000 and filling up name, address, and telephone number. A member can cancel his membership and take back his deposit, if he has no dues outstanding against him.
- Whenever the store purchases a new item, the details such as date of procurement and price are entered. The daily rental charge is also entered by the manager. After passage

of a year, the daily rental charge is automatically halved.

- A member can take on loan at most one video CD and one music CD each time. The details are entered by a store clerk and a receipt indicating the daily rental charge is printed.
- Whenever a member returns his loaned item, the amount to be paid is displayed. After the amount is paid, the items are marked returned.
- If a customer loses or damages any item, the full price of the item is charged to him and the item is removed from the inventory.
- If an item lies unissued for more than a year, it is sold to the members at 10 per cent of the purchase price and the item is removed from the inventory.
- The manager can at any time check the profit/loss account.

43. Develop the SA/SD diagrams for the following **Elevator Controller** software.

The controller software of the elevator gets inputs from lift users through the push button switches mounted inside the elevator and near the lift door at each floor. The controller generates output by giving commands to the motor controller and the lift door controller. At each floor, only two switches are installed. The switches are marked with up and down arrows indicating the request to go either in the up or down direction. Inside the lift, there is a panel of switches, with one switch labelled for each floor. Also there is an emergency stop switch. A user at a floor can request for the lift and indicate the required direction of travel by pressing the appropriate button. The requests for the lift arriving from various floors are queued by the controller and it serves the request in a shortest distance first manner, if the lift is idle. Once the lift stops at a floor, the user can press a switch labelled with the floor number to request the lift to go to that floor. After the floor request button is pressed, the lift controller times out after one minute and then starts to close the lift door. The successful closing of the lift door is indicated by the signals generated by a contact sensor. The lift starts to move in the required direction after the lift doors have completely closed. A user inside the lift can stop the lift doors from closing by pressing the emergency stop switch before the lift starts to move. Once the lift starts to move, pressing the emergency stop switch has no effect. At each

floor, there is a touch sensor that indicates to the controller that a lift has reached the floor and the controller commands the motor to stop after a required floor is reached. After 30 seconds of reaching a floor, the lift door is opened by issuing a command to the door controller. If there is a power failure any time during a lift's movement, the lift reverts to a safe mode in which it shuts down the motor and a mechanical backup arrangement slides the lift to the ground floor and the manual door opening handle is enabled, which the user can use to open the lift doors.

Chapter 7

OBJECT MODELLING USING UML

In recent years, the object-oriented software development style has become very popular and is at present being widely used in industry as well as in academic circles. Since its inception in the early eighties, the object technology has made rapid progress. From a modest beginning in the early eighties, the advancements to the object technology gathered momentum in the nineties and the technology is now nearing maturity. Considering the widespread use and popularity of the object technology in both industry and academia, it is important to learn this technology well.

It is well known that mastering an object-oriented programming language such as Java or C++ rarely equips one with the skills necessary to develop good quality object-oriented software—it is important to learn the object-oriented design skills well. Once a good design has been arrived at, it is easy to code it using an object-oriented language. It has now even become possible to automatically generate much of the code from the design by using a CASE tool. In order to arrive at a satisfactory object-oriented design (OOD) solution to a problem, it is necessary to create several types of models. But, one may ask: "What has modelling got anything to do with designing?" Let us answer this question in the following:

A model is constructed by focusing only on a few aspects of the problem and ignoring the rest. The model of a problem is called an analysis model. On the other hand, the model of the solution (code) is called the design model. The design model is usually obtained by carrying out iterative refinements to the analysis model using a design methodology.

Note that any design is a model of the solution, whereas any model of the problem is an analysis model. In this chapter, we shall discuss how to document a model using a modelling language. In the subsequent chapter, we shall discuss a design process that can be used to iteratively refine an

analysis model into a design model.

In the context of model construction, we need to carefully understand the distinction between a modelling language and a design process, since we shall use these two terms frequently in our discussions.

Modelling language: A modelling language consists of a set of notations using which design and analysis models are documented.

Design process: A design process addresses the following issue: "Given a problem description, how to systematically work out the design solution to the problem?" In other words, a design process consists of a step by step procedure (or recipe) using which a problem description can be converted into a design solution. A design process is, at times, also referred to as a design methodology. In this text, we shall use the terms design process and design methodology interchangeably.

A model can be documented using a modelling language such as unified modelling language (UML). Over the last decade, UML has become immensely popular. UML has also been accepted by ISO as a standard for modelling object-oriented systems. In this Chapter, we primarily discuss the syntax and semantics of UML. However, before discussing the nitty- gritty of the syntax and semantics of UML, we review a few basic concepts and terminologies that have come to be associated with object-orientation.

7.1 BASIC OBJECT-ORIENTATION CONCEPTS

The principles of object-orientation have been founded on a few simple concepts. These concepts are pictorially shown in Figure 7.1. After discussing these basic concepts, we examine a few related technical terms.

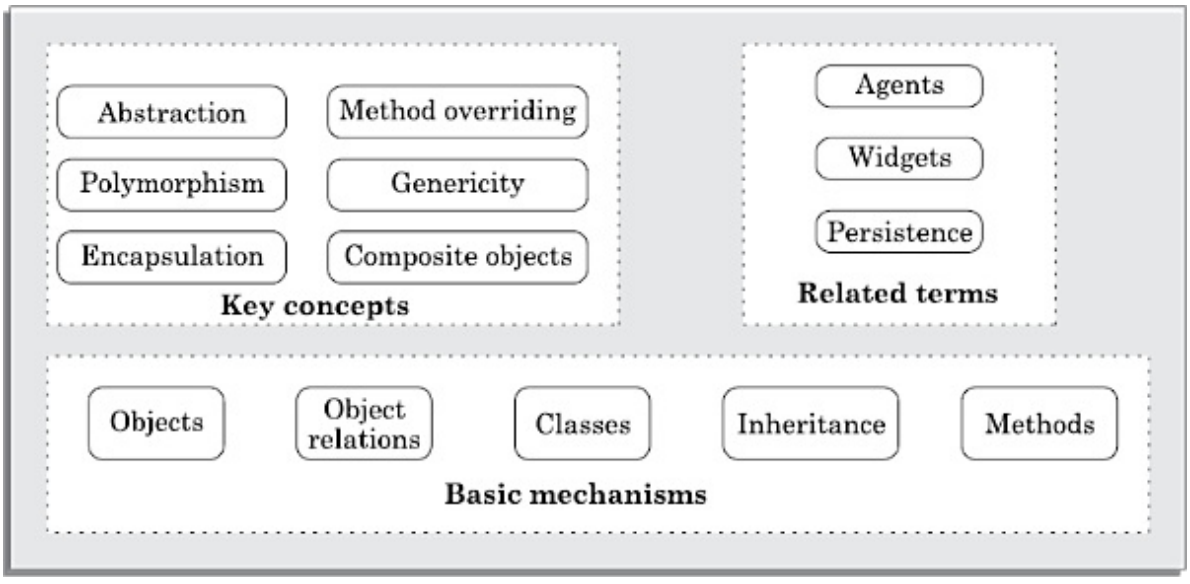

Figure 7.1: Important concepts used in the object-oriented approach.

7.1.1 Basic Concepts

A few important concepts that form the corner stones of the object-oriented paradigm have pictorially been shown in Figure 7.1. In the following sections and subsections, we discuss these concepts in more detail.

Objects

In the object-oriented approach, it is convenient to imagine the working of a software in terms of a set of interacting objects. This is analogous to the way object manipulations take place in a real-world manual system for getting some work done. For example, consider a manually operated library system. For issuing a book, an issue register needs to be filled up and then the return date needs to be stamped on the book. In an object-oriented library automation software, analogous activities involving the book object and the issue register object take place.

Each object in an object-oriented program usually represents a tangible real-world entity such as a library member, a book, an issue register, etc. However while solving a problem, it becomes advantageous at times to consider certain conceptual entities (e.g., a scheduler, a controller, etc.) as objects as well. This simplifies the solution and helps to arrive at a good design.

A key advantage of considering a system as a set of objects is the following:

When the system is analysed, developed, and implemented in terms of objects, it becomes easy to understand the design and the implementation of the system, since objects provide an excellent decomposition of a large problem into small parts.

Each object essentially consists of some data that is private to the object and a set of functions (termed as operations or methods) that operate on those data. This aspect has pictorially been illustrated in Figure 7.2. Observe that the data of the object can only be accessed by the methods of the object. Consequently, the only access point to the data for the external objects is through the invocation of the methods of the object. In fact, the methods of an object have the sole authority to operate on the data that is private to the object. In other words, no object can directly access the data of any other object.Therefore, each object can be thought of as hiding its internal data from other objects. However, an object can access the private

data of another object by invoking the methods supported by that object. This mechanism of hiding data from other objects is popularly known as the principle of data hiding or data abstraction. Data hiding promotes high cohesion and low coupling among objects, and therefore is considered to be an important principle that can help one to arrive at a reasonably good design.

As already mentioned, each object stores some data and supports certain operations on the stored data. As an example, consider the libraryMember object of a library automation application. The private data of a libraryMember object can be the following:

- name of the member
- membership number
- address
- phone number
- e-mail address
- date when admitted as a member
- membership expiry date
- books outstanding

The operations supported by a libraryMember object can be the following:

- issue-book
- find-books-outstanding
- find-books-overdue
- return-book
- find-membership-details

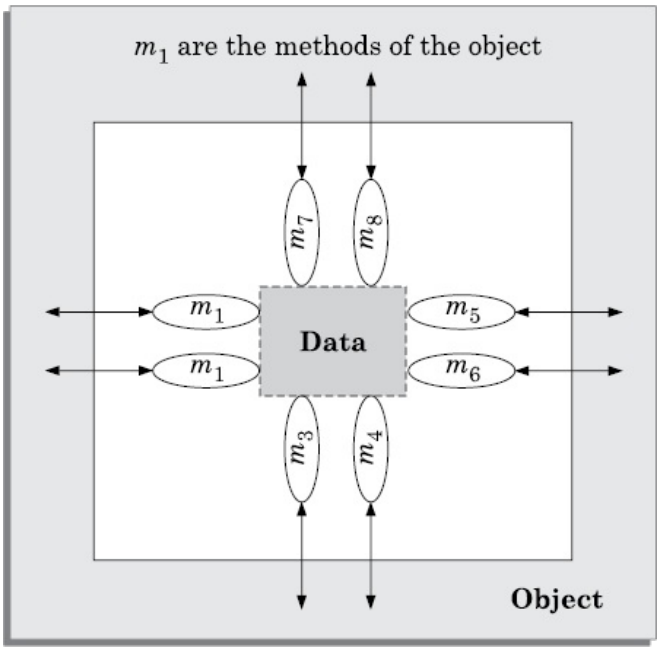

Figure 7.2: A model of an object

The data stored internally in an object are called its attributes, and the operations supported by an object are called its methods.

Though the terminologies associated with object-orientation are simple and well-defined, a word of caution here: the term 'object' is often used rather loosely in the literature and also in this text. Often an 'object' would mean a single entity. However, at other times, we shall use it to refer to a group of similar objects (class). In this text, usually the context of use would resolve the ambiguity, if any.

Is it really true that an object-oriented program consists of objects only?

So far, we claimed that an object-oriented program consists of objects only and the program execution involves through interaction of these objects. However, a program written using in the present programming languages such as Java, C++, etc. work through interaction of objects and primitive data. These programming languages consider that primitive data are not objects and distinctions are made between these in several ways. To get perspective on this, we need to examine a bit of programming history.

The first object-oriented programming language was Smalltalk. It was developed in the 1970s at the Xerox research center in USA. It was a pure object-oriented language in the sense that applications developed in this language consisted of objects only. In other words, no distinction was made

between objects and instances of primitive types of data (int, float, etc.). This was intended to gain better programmer acceptance. Small talk did not find widespread acceptance among the programmers, since it required the programmers to learn and use an entirely different program development paradigm. However, in the later generation programming languages such as C++ and Java, instances of primitive data types were treated differently from objects. For example, in these languages, objects are passed as reference arguments to methods, whereas primitive data types are passed by value. The motivation behind this distinction was to make the object-oriented languages appear as a small extension to procedural languages rather than bringing in any radically new approach. In this light, C++ was designed to retain the procedural approach of C and only extend it with the object-orientation constructs. Of course, different object-oriented constructs were translated into C code by a preprocessor. This made it easy for the procedural programmers to migrate to C++. This was possibly an important reason why C++ gained much more popularity compared to a pure object-oriented programming language. At present, object-oriented programming languages such as Java and C++ distinguish primitive data types from objects and treat them very differently.

Class

Similar objects constitute a class. That is, all the objects constituting a class possess similar attributes and methods. For example, the set of all library members would constitute the class LibraryMember in a library automation application. In this case, each library member object has attributes such as `member name`, `membership number`, `member address`, etc. and also has methods such as `issue-book`, `return-book`, etc. Once we define a class, it can be used as a template for object creation.

Let us now investigate the important question as to whether a class is an abstract data type (ADT). To be able to answer this question, we must first be aware of the basic definition of an ADT. We first discuss the same in a nutshell. There are two things that are inherent to an ADT—abstract data and data type. In programming language theory, a data type identifies a group of variables having a particular behaviour. A data type can be instantiated to create a variable. For example, `int` is a type in C++ language. When we write an instruction int i; we actually create an instance of `int`. From this, we can infer the following—An ADT is a type where the data contained in each

instantiated entity is abstracted (hidden) from other entities. Let us now examine whether a class supports the two mechanisms of an ADT.

Abstract data: The data of an object can be accessed only through its methods. In other words, the exact way data is stored internally (stack, array, queue, etc.) in the object is abstracted out (not known to the other objects).

Data type: In programming language terminology, a data type defines a collection of data values and a set of predefined operations on those values. Thus, a data type can be instantiated to create a variable of that type. We can instantiate a class into objects. Therefore, a class is a data type.

It can be inferred from the above discussions that a class is an ADT. But, an ADT need not be a class, since to be a class they need to support the inheritance and other object-orientation properties.

Methods

The operations (such as `create`, `issue`, `return`, etc.) supported by an object are implemented in the form of methods. Notice that we are distinguishing between the terms operation and method. Though the terms 'operation' and 'method' are sometimes used interchangeably, there is a technical difference between these two terms which we explain in the following.

An operation is a specific responsibility of a class, and the responsibility is implemented using a method. However, it is at times useful to have multiple methods to implement a responsibility. In this case, all the methods share the same name (that is, the name of the operation), but parameter list of each method is required to be different for enabling the compiler to determine the exact method to be bound on a method call. We therefore say that in this case, the method name is overloaded with multiple implementations of the operation. As an example, consider the following. Suppose one of the responsibilities of a class named Circle is to create instances of itself. Assume that the class provides three definitions for the create operation—`int create()`, `int create(int radius)` and `int create(float x, float y, int radius)`;. In this case, we say that create is an overloaded method.

The implementation of a responsibility of a class through multiple methods with the same method name is called method overloading.

Methods are the only means available to other objects in a software for accessing and manipulating the data of another object. The set of valid

messages to an object constitutes its protocol. Let us now try to understand the distinction between a message and a method.

In Smalltalk, an object could request the services (i.e., invoke methods) of other objects by sending messages to them. The idea was that the mechanism of message passing would lead to weak coupling among objects. Though this was an important feature of Smalltalk, yet the programmers who were trying to migrate from procedural programming to object-oriented programming, found it to be a paradigm shift and therefore difficult to accept. Subsequently, C++ implemented message passing by method invocation (similar to a function call). This was rapidly accepted by the programmers. Later object-oriented languages such as Java have followed the same trait of retaining the method invocation feature, that is normally associated with the procedural languages.

7.1.2 Class Relationships

Classes in a programming solution can be related to each other in the following four ways:

- Inheritance
- Association and link
- Aggregation and composition
- Dependency

In the following subsection, we discuss these different types of relationships that can exist among classes.

Inheritance

The inheritance feature allows one to define a new class by incrementally extending the features of an existing class. The original class is called the base class (also called superclass or parent class) and the new class obtained through inheritance is called the derived class (also called a subclass or a child class). The derived class is said to inherit the features of the base class. An example of inheritance has been shown in Figure 7.3. In Figure 7.3, observe that the classes `Faculty`, `Students`, and `Staff` have been derived from the base class `LibraryMember` through an inheritance relationship (note the special type of arrow that has been used to draw it). The inheritance relation between library member and faculty can alternatively be expressed as the following—A faculty is a type of library member. So, the inheritance relationship is also at times called is a relation.

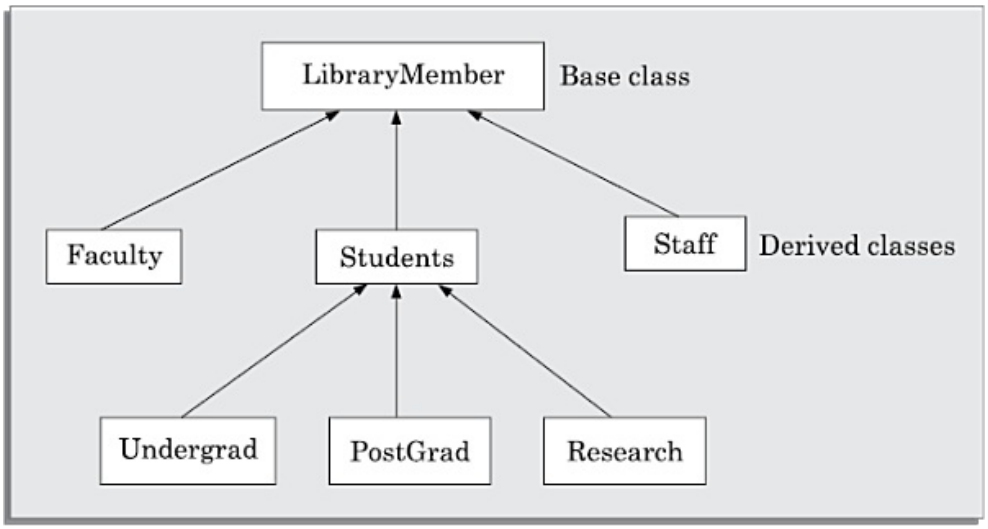

Figure 7.3: Library information system example.

A base class is said to be a generalisation of its derived classes. This means that the base class should contain only those properties (i.e., data and methods) that are common to all its derived classes. For example, in Figure 7.3 each of the derived classes supports the issue-book method, and this method is supported by the base class as well. In other words, each derived class is a specialised base class that extends the base class functionalities in certain ways.

> Each derived class can be considered as a specialisation of its base class because it modifies or extends the basic properties of the base class in certain ways. Therefore, the inheritance relationship can be viewed as a generalisation-specialisation relationship.

Observe that in Figure 7.3 the classes `Faculty`, `Student`, and `Staff` are all special types of library members. Several things are common among the members. These include attributes such as membership id, member name and address, date on which books issued out, etc. However, for the different categories of members, the issue procedure differs since different types members are issued books for different durations. We can convey as much by saying that the classes `Faculty`, `Staff`, and `Students` are special types of `LibraryMember` classes. Using the inheritance relationship, different classes can be arranged in a class hierarchy (or class tree) as shown in Figure 7.3.

In addition to inheriting all the data and methods of the base class, a derived class usually defines some new data and methods. A derived class may even redefine a method which already exists in the base class. Redefinition of a method which already exists in its base class is termed as

method overriding.

> When a new definition of a method that existed in the base class is provided in a derived class, the method is said to be overridden in the derived class.

The inheritance relationship existing among certain classes in a library automation system is shown in Figure 7.3. As shown, `LibraryMember` is the base class for the derived classes `Faculty, Student, and Staff`. Similarly, `Student` is the base class for the derived classes `Undergrad, Postgrad,` and `Research`. Each derived class inherits all the data and methods of the base class, and defines some additional data and methods or modifies some of the inherited data and methods. The inheritance relationship has been represented in Figure 7.3 using a directed arrow drawn from a derived class to its base class. We now illustrate how the method of a base class is overridden by the derived classes. In Figure 7.3, the base class `LibraryMember` might define the following data—`member name, address, and library membership number`. Though faculty, student, and staff classes inherit these data, they need to redefine their respective issueBook methods because for the specific library that we are modelling, the number of books that can be borrowed and the duration of loan are different for different categories of library members.

Inheritance is a basic mechanism that every object-oriented programming language needs to support. If a language supports ADTs, but does not support inheritance, then it is called an object-based language and not object-oriented. An example of an object-based programming language is Ada.

Now let us try to understand why do we need the inheritance relationship in the first place. Can't we program as well without using the inheritance relationship?

> Two important advantages of using the inheritance mechanism in programming include code reuse and simplicity of program design.

Let us examine how code reuse and simplicity of design come about while using the inheritance mechanism. If certain methods or data are present in several classes, then instead of defining these methods and data in each of the classes separately, these methods and data are defined only once in the base class and then inherited by each of its subclasses. For example, in the Library Information System example of Figure 7.4, each category of member (that is, `Faculty, Student,` and `Staff`) need to store the following data

—`member-name`, `member-address`, and `membership-number`. Therefore, these data are defined in the base class `LibraryMember` and are inherited by all it's subclasses. Another advantage that accrues from the use of the inheritance mechanism is the conceptual simplification brought about through the reduction of the number of independent features of the different classes and incremental understanding of the different classes becomes possible. Thus, inheritance can be considered as a use of the abstraction mechanism we discussed in Chapter 1. The class at the root of an inheritance hierarchy (e.g. LibraryMember in Figure 7.3) is the simplest to understand—as it has the least number of data and method members compared to all other classes in the hierarchy. The classes at the leaf-level of the inheritance hierarchy have the maximum number of features (data and method members) because they inherit features of all their ancestors, and therefore turn out to be the toughest to understand.

In a large class hierarchy, it is easier to first understand the root class and then to recursively understand its children classes in the hierarchy until the leaf level classes are reached.

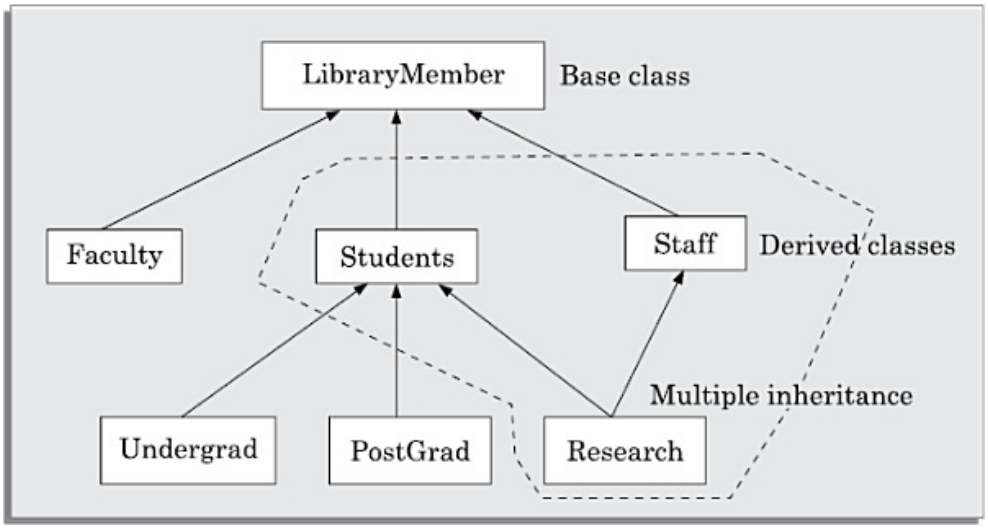

Figure 7.4: An example of multiple inheritance.

Multiple inheritance

Construction of the class relationships for a given problem consists of identifying and representing four types of relations—inheritance, composition/aggregation, association, and dependency. However, at times it may so happen that some features of a class are similar to one class and a few other features of the class are similar to those of another class. In this case, it would be useful if the class could be allowed to inherit features from both the classes. Using the multiple inheritance feature, a class can inherit

features from multiple base classes.

> Multiple inheritance is a mechanism by which a subclass can inherit attributes and methods from more than one base class.

Consider the following example of a class that is derived from two base classes through the use of the multiple inheritance mechanism. In this example, in an academic institute research students can also be employed as staff of the institute, then some of the characteristics of the `Research` class are similar to the `Student` class (e.g., every student would have a roll number) while some other characteristics (e.g., having a basic salary and employee number, etc.) might be similar to the `Staff` class. Using multiple inheritance the class `Research` can inherit features from both the classes `Student` and `Staff`. In Figure 7.4, we have shown the class `Research` to be derived from both the `Student` and `Staff` classes by drawing inheritance arrows to both the parent classes of the `Research` class.

Association and link

Association is a common type of relation among classes. When two classes are associated, they can take each others help (i.e. invoke each others methods) to serve user requests. More technically, we can say that if one class is associated with another bidirectionally, then the corresponding objects of the two classes know each others ids (identities). As a result, it becomes possible for the object of one class to invoke the methods of the corresponding object of the other class.

Consider the following example. A `Student can register in one Elective subject`. In this example, the class `Student` is associated with the class `ElectiveSubject`. Therefore, an `ElectiveSubject` object (e.g. MachineLearning) would know the ids of all `Student` objects that have registered for the Subject and can invoke their methods such as `printName`, `printRoll` and `enterGrade`. This relationship has been represented in Figure 6(a). When an object knows some other object, it must internally store its id. For example, for an object of `ElectiveSubject` class e1 to invoke the `printName()` method one of the registered students s1 , it must execute the code s1 `.printName()`. Thus, it should have stored the id s1 of the registered student as an attribute. The association relationship can either be bidirectional or unidirectional. That is, both the associated classes know each other (store each others ids). We have graphically shown this

association between Student class and `ElectiveSubject` in Figure 7.5(a).

Consider another example of association between two classes: `Library Member` borrows Books. Here, borrows is the association between the class LibraryMember and the class Book. The association relation would imply that given a book, it would be possible to determine the borrower and vice versa.

n-ary association

Binary association between classes is very commonly encountered in design problems. However, there can be situations where three or more different classes can be involved in an association. As an example of a ternary association, consider the following—A person books a ticket for a certain show. Here, an association exists among the classes Person, Ticket, and Show. This example of ternary association relationship has pictorially been shown in Figure 7.5(b).

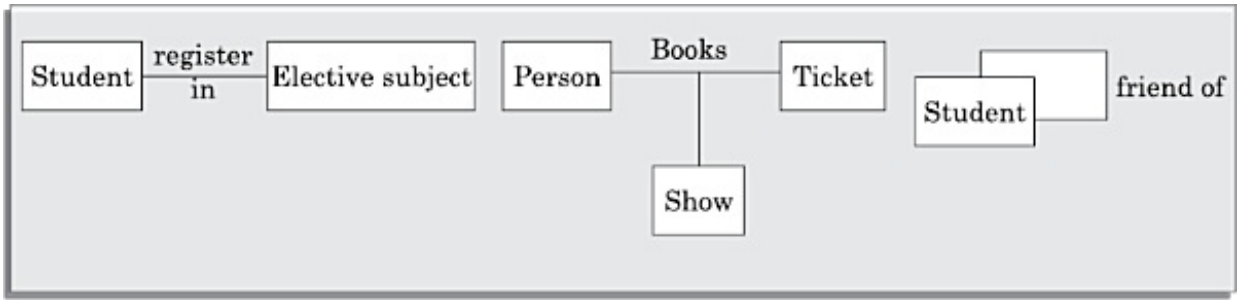

Figure 7.5: Example of (a) binary (b) ternary (c) unary association.

A class can have an association relationship with itself. This is called recursive association or unary association. As an example, consider the following—two students may be friends. Here, an association named friendship exists among pairs of objects of the `Student` class. This has pictorially been shown in Figure 7.5(c).

> In unary association, two (or more) different objects of the same class are linked by the association relationship.

When two classes are associated, the relationship between two objects of the corresponding classes is called a link.

> An association describes a group of similar links. Alternatively, we can say that a link can be considered as an instance of an association relation. Let us now try to identify the association relation from the text description of a problem.

Example 7.1 Consider the following extract from a problem description. Identify the association relation among classes and the corresponding

association links among objects from an analysis of the description. "A person works for a company. Ram works for Infosys. Hari works for TCS."

Answer: In this example, an association relationship named works for exists between the classes Person and Company. Ram works for Infosys, this implies that a link exists between the object Ram and the object Infosys. Similarly, a works for link exists between the objects Hari and TCS.

During a run of the system, new links can get formed among the objects of the associated classes and some existing links may get dissolved. Note that some objects may not have any association link to any of the objects of the associated class. Some objects of the associated classes may have not have links. For example in course of time, Ram may resign from Infosys and join Wipro. In this case, the link between Ram and Infosys breaks and a link between Ram and Wipro gets formed. But, suppose Ram does not join any other job after leaving Infosys. In this case, Ram does not have works for link with any Company object. Thus, links are time varying (or dynamic) in nature. Association relationship between two classes is static in nature.

> If two classes are associated, then the association relationship exists at all points of time. In contrast, links between objects are dynamic in nature. Links between the objects of the associated classes can get formed and dissolved as the program executes.

An association between two classes simply means that zero or more links may be present among the objects of the associated classes at any time during execution.

Mathematically, a link can be considered to be a tuple. Consider the following example. "Amit has borrowed the book Graph Theory." Here, a link named borrowed has got established between the objects Amit and the Graph Theory book. This link can also be expressed as the ordered pair of object instances {Amit,Graph Theory}.

Composition and aggregation

Composition and aggregation represent part/whole relationships among objects. Objects which contain other objects are called composite objects. As an example, consider the following—`A Book object can have upto ten Chapters`. In this case, a Book object is said to be composed of upto ten `Chapter` objects. The composition/aggregation relationship can also be read as follows—`A Book` has upto ten `Chapter` objects (shown in Figure

7.6). For this reason, the composition/aggregation relationship is also known as has a relationship. Aggregation/composition can occur in a hierarchy of levels. That is, an object contained in another object may itself contain some other object. Composition and aggregation relationships cannot be reflexive. That is, an object cannot contain an object of the same type as itself.

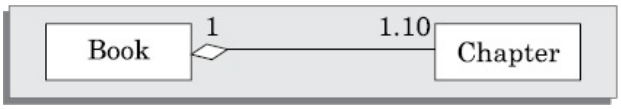

Figure 7.6: Example of aggregation relationship.

Dependency

A class is said to be dependent on another class, if any changes to the latter class necessitates a change to be made to the dependent class.

A dependency relation between two classes shows that any change made to the independent class would require the corresponding change to be made to the dependent class.

Dependencies among classes may arise due to various causes. Two important reasons for dependency to exist between two classes are the following:

- A method of a class takes an object of another class as an argument.
- A class implements an interface class. In this case, dependency arises due to the following reason. If some properties of the interface class are changed, then a change becomes necessary to the class implementing the interface class as well.

Abstract class

Classes that are not intended to produce instances of themselves are called abstract classes. In other words, an abstract class cannot be instantiated. If an abstract class cannot be instantiated to create objects, then what is the use of defining an abstract class? Abstract classes merely exist so that behaviour common to a variety of classes can be factored into one common location, where they can be defined once. Definition of an abstract class helps to push reusable code up in the class hierarchy, thereby enhancing code reuse.

By using abstract classes, code reuse can be enhanced and the effort required to develop software brought down.

Abstract classes usually support generic methods. These methods help to standardise the method names and input and output parameters in the derived classes. The subclasses of the abstract classes are expected to provide the concrete implementations for these methods. For example, in a library automation software `Issuable` can be an abstract class (see Figure 7.7) and the concrete classes `Book`, `Journal`, and `CD` are derived from the `Issuable` class. The `Issuable` class may define several generic methods such as `issue`. Since the issue procedures for books, journals, and CDs would be different, the `issue` method would have to be overridden in the `Book`, `Journal`, and `CD` classes. Though this does not help in code reuse, but helps to have standardised implementations of the `issue` method across different concrete classes. Observe that `Issuable` is an abstract class and cannot be instantiated. On the other hand, `Book`, `Journal`, and `CD` are concrete classes and can be instantiated to create objects.

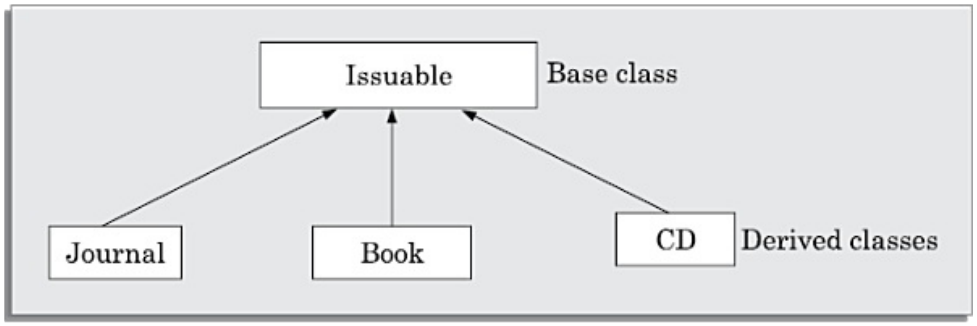

Figure 7.7: An example of an abstract class.

7.1.3 How to Identify Class Relationships?

Suppose we want to write the code for a simple programming problem. How do we identify the classes and their relationships from this description, so that we can write the necessary code? This can be done by a careful analysis of the sentences given in the problem description. The nouns in a sentence often denote the classes. On the other hand, the relationships among classes are usually indicated by the presence of certain key words. In the following, we give examples of a few key words (shown in italics) that indicate the specific relationships among two classes A and B:

Composition

- B is a permanent part of A
- A is made up of Bs

- A is a permanent collection of Bs

Aggregation

- B is a part of A
- A contains B
- A is a collection of Bs

Inheritance

- A is a kind of B
- A is a specialisation of B
- A behaves like B

Association

- A delegates to B
- A needs help from B
- A collaborates with B. Here collaborates with can be any of a large variety of collaborations that are possible among classes such as employs, credits, precedes, succeeds, etc.

7.1.4 Other Key Concepts

We now discuss a few other key concepts used in the object-oriented program development approaches:

Abstraction

Let us first recapitulate how the abstraction mechanism works (we had already discussed this basic mechanism in Section 1.3.2). Abstraction is the selective examination of certain aspects of a problem while ignoring all the remaining aspects of a problem. In other words, the main purpose of using the abstraction mechanism is to consider only those aspects of the problem that are relevant to a given purpose and to suppress all aspects of the problem that are not relevant.

> The abstraction mechanism allows us to represent a problem in a simpler way by considering only those aspects that are relevant to some purpose and omitting all other details that are irrelevant.

Many different abstractions of the same problem can be constructed depending on the purpose for which the abstractions are made. The abstraction mechanism cannot only help the development engineers to understand and appreciate a problem better while working out a solution, but can also help better comprehension of a system design by the maintenance team. Abstraction is supported in two different ways in an object-oriented designs (OODs). These are the following:

Feature abstraction: A class hierarchy can be viewed as defining several levels (hierarchy) of abstraction, where each class is an abstraction of its subclasses. That is, every class is a simplified (abstract) representation of its derived classes and retains only those features that are common to all its children classes and ignores the rest of the features. Thus, the inheritance mechanism can be thought of as providing feature abstraction.

Data abstraction: An object itself can be considered as a data abstraction entity, because it abstracts out the exact way in which it stores its various private data items and it merely provides a set of methods to other objects to access and manipulate these data items. In other words, we can say that data abstraction implies that each object hides (abstracts away) from other objects the exact way in which it stores its internal information. This helps in developing good quality programs, as it causes objects to have low coupling with each other, since they do not directly access any data belonging to each other. Each object only provides a set of methods, which other objects can use for accessing and manipulating this private information of the object. For example, a stack object might store its internal data either in the form of an array of values or in the form of a linked list. Other objects would not know how exactly this object has stored its data (i.e. data is abstracted) and how it internally manipulates its data. What they would know is the set of methods such as push, pop, and top-of-stack that it provides to the other objects for accessing and manipulating the data.

An important advantage of the principle of data abstraction is that it reduces coupling among various objects, Therefore, it leads to a reduction of the overall complexity of a design, and helps in easy maintenance and code reuse.

Abstraction is a powerful mechanism for reducing the perceived complexity of software designs. Analysis of the data collected from several software development projects shows that software productivity is inversely proportional to the perceived software complexity. Therefore, implicit use of abstraction, as it takes place in object-oriented development, is a promising

way of increasing productivity of the software developers.

Encapsulation

The data of an object is encapsulated within its methods. To access the data internal to an object, other objects have to invoke its methods, and cannot directly access the data. This concept is schematically shown in Figure 7.8. Observe from Figure 7.8 that there is no way for an object to access the data private to another object, other than by invoking its methods. Encapsulation offers the following three important advantages:

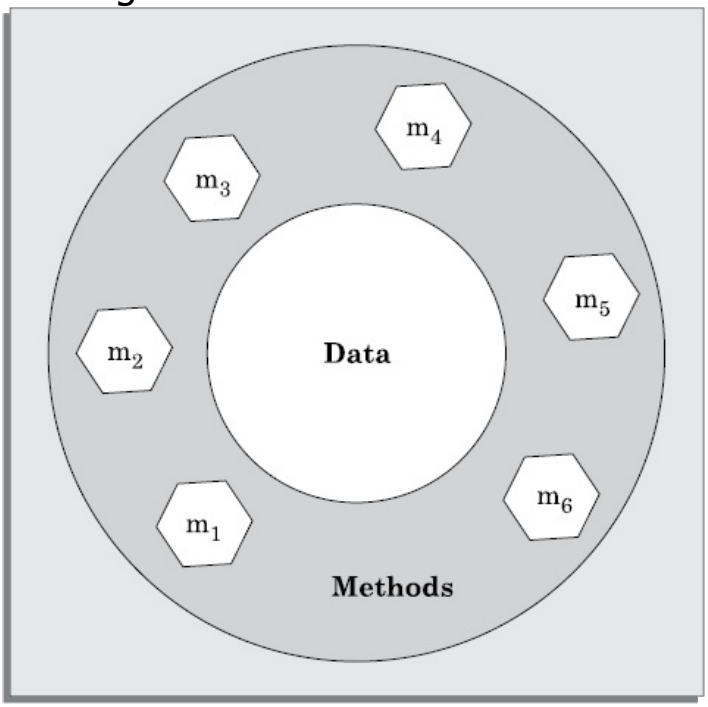

Figure 7.8: Schematic representation of the concept of encapsulation.

Protection from unauthorised data access: The encapsulation feature protects an object's variables from being accidentally corrupted by other objects. This protection includes protection from unauthorised access and also protection from the problems that arise from concurrent access to data such as deadlock and inconsistent values.

Data hiding: Encapsulation implies that the internal structure data of an object are hidden, so that all interactions with the object are simple and standardised. This facilitates reuse of a class across different projects. Furthermore, if the internal data or the method body of a class are modified, other classes are not affected. This leads to easier maintenance and bug correction.

Weak coupling: Since objects do not directly change each others internal

data, they are weakly coupled. Weak coupling among objects enhances understandability of the design since each object can be studied and understood in isolation from other objects.

Polymorphism

Polymorphism literally means poly (many) morphism (forms). Remember that in Chemistry, diamond, graphite, and coal are called polymorphic forms of carbon. That is, though diamond, coal, and graphite are essentially carbon, they behave very differently. In an analogous manner in the object-oriented paradigm, polymorphism denotes that an object may respond (behave) very differently even when the same operation is invoked on it depending on the exact polymorphic object to which the call gets bound.

There are two main types of polymorphisms in object-orientation:

Static polymorphism: Static polymorphism occurs when multiple methods implement the same operation. In this type of polymorphism, when a method is called (same method name but different parameter types), different behaviour (actions) would be observed. This type of polymorphism is also referred to as static binding, because the exact method to be bound on a method call is determined at compiled-time (statically). Let us try to understand static binding through the following example. Suppose a class named Circle has three definitions for the create operation—`int create()`, `int create(int radius)`, and `int create(float x, float y, int radius)`. Recollect that when multiple methods implement the same operation, then the mechanism used is called method overloading. (Notice the difference between an operation and a method.) When the same operation (e.g. create) is implemented by multiple methods, the method name is said to be overloaded. One definition of the create operation does not take any argument `(create())` and creates a circle with default parameters. The second definition takes the center point and the radius radius of the circle as its parameters `(create(float x, float y, float radius)`. Assume that in both the above cases, the fill style would be set to the default value "no fill". The third definition of the create operation takes the center point, the radius, and a fill style as its input. When the create method is invoked, depending on the parameters given in the invocation, the matching method can be easily determined during compilation by examining the parameter list of the call and the call would get be statically bound. If `create` method is invoked with no parameters, then a default circle

would be created. If only the center and radius are supplied, then an appropriate circle would be created with no fill type, and so on. The definition of the `Circle` class with the overloaded `create` method is shown in Figure 7.9. **Dynamic polymorphism:** Dynamic polymorphism is also called dynamic binding. In dynamic binding,the exact method that would be invoked (bound) on a method call can only be known at the run time (dynamically) and cannot be determined at compile time. That is, the exact behaviour that would be produced on a method call cannot be predicted at compile time and can only be observed at run time.

```
class Circle{
        private float x, y, radius;
        private int fillType;

        public create();
        public create(float x, float y, float radius);
        public create(float x, float y, float radius, int fillType);
    }
```

Figure 7.9: Circle class with overloaded create method.

Let us now explain how dynamic binding works in object-oriented programs. Dynamic binding is based on two important concepts:

- Assignment of an object to another compatible object.
- Method overriding in a class hierarchy.

Assignment to compatible of objects

In object-orientation, objects of the derived classes are compatible with the objects of the base class. That is, an object of the derived class can be assigned to an object of the base class, but not vice versa. Also an object cannot be assigned to an object of a sibling class or an object of a totally unrelated class for obvious reasons. This is an important principle in object-orientation and is known as the Liskov Substitution principle. To understand this principle, recollect that a derived class defines some extra attributes and assignment of an object of the base class to an object of the derived class can leave those attributes undefined.

Method overriding

We have already explained the method overriding principle in which a

derived class provides a new definition to a method of the base class.

Let us now understand how dynamic binding works by making use of the above two mechanisms. Suppose we have defined a class hierarchy of different geometric shapes for a graphical drawing package as shown in Figure 7.10. As can be seen from the figure, `Shape` is an abstract class and the classes `Circle`, `Rectangle`, and `Line` are directly derived from it and further the classes `Ellipse`, `Square` and `Cube` have been derived to form an inheritance hierarchy. Now, suppose the `draw` method is declared in the `Shape` class and is overridden in each derived class. Further, suppose a set of different types of `Shape` objects have been created one by one. By Liskov's substitution principle, the created `Shape` objects can be stored in an array of type Shape. If the different types of geometric objects making up a drawing are stored in an array of type `Shape`, then a call to the `draw` method for each object would take care to display the appropriate drawing element. That is, the same `draw` call to a Shape object would take care of drawing the appropriate drawing object. Observe that due to dynamic binding, a call to the `draw` method of the shape class takes care of displaying the appropriate drawing object residing in the shape array. This is illustrated in the code segment shown in Figure 7.11.

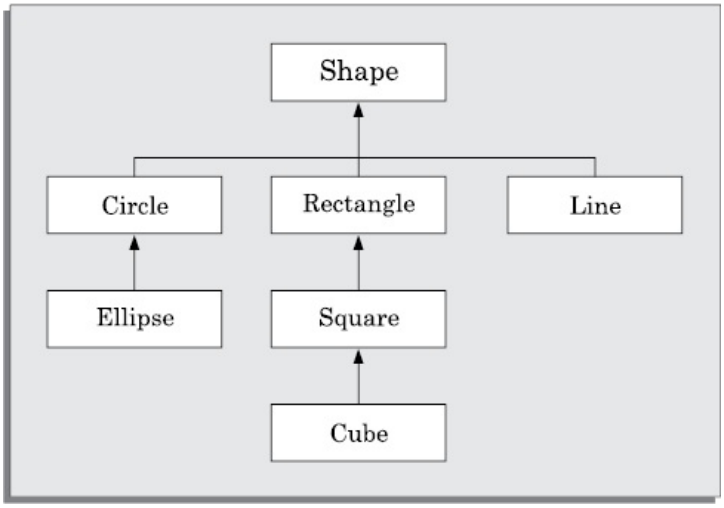

Figure 7.10: Class hierarchy of geometric objects.

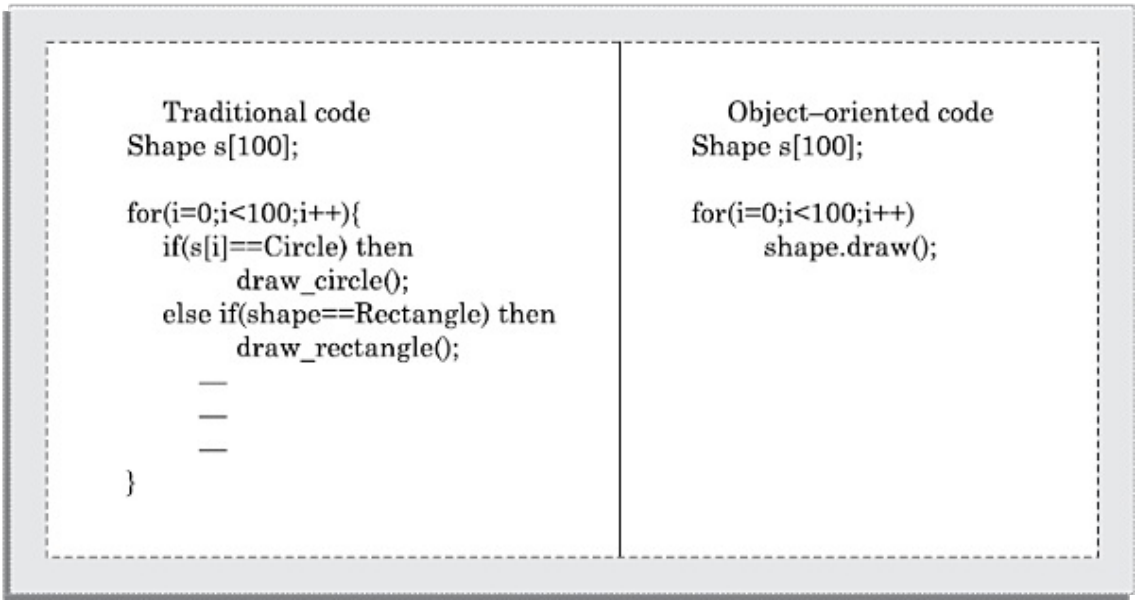

Figure 7.11: Traditional code versus object-oriented code incorporating the dynamic binding feature.

We analyse the advantage of polymorphism by comparing the code segments of an object-oriented program and a traditional program for drawing various graphic objects on the screen (shown in Figure 7.11). Using dynamic binding, a programmer can invoke the generic method of an object and leave the exact way in which this message would be handled would be decided‘ depending on the object that is currently assigned to the receiving object. With dynamic binding, new derived objects can be added with minimal changes to a program.

The principal advantage of dynamic binding is that it leads to elegant programming and facilitates code reuse and maintenance.

It can be easily seen from Figure 7.11 that the use of dynamic binding, the object-oriented code is much more concise, understandable, and intellectually appealing as compared to equivalent procedural code. Further, suppose that in the example program segment, it is later found necessary to handle a new graphics drawing primitive, say ellipse. Then, the procedural code has to be changed by adding a new if-then-else clause. However, in case of an object-oriented program, the code need not change, only a new class called Ellipse has to be derived in the Shape hierarchy.

We can now summarise the mechanism of dynamic binding as follows:

Even when the method of an object of the base class is invoked, an appropriate overridden method of a derived class would be invoked depending on the exact object that may have been assigned at the run-time to the object of the base class.

Genericity

Genericity is the ability to parameterise class definitions. For example, while defining a class stack of different types of elements such as integer stack, character stack, floating-point stack, etc.; genericity permits us to define a generic class of type stack and later instantiate it either as an integer stack, a character stack, or a floating-point stack as may be required. This can be achieved by assigning a suitable value to a parameter used in the generic class definition.

7.1.5 Related Technical Terms

In the following, we discuss a few terms related to object-orientation as follows:

Persistence

Objects usually get destroyed once a program completes its execution. Persistent objects are stored permanently. That is, they live across different executions. An object can be made persistent by maintaining copies of the object in a secondary storage or in a database.

Agents

A passive object is one that performs some action only when requested through invocation of some of its methods. An agent (also called an active object), on the other hand, monitors events occurring in the application and takes actions autonomously. Agents are used in applications such as monitoring exceptions. For example, in a database application such as accounting, an agent may monitor the balance sheet and would alert the user whenever inconsistencies arise in a balance sheet due to some improper transaction taking place.

Widget

The term widget stands for window object. A widget is a primitive object used for graphical user interface (GUI) design. More complex graphical user interface design primitives (widgets) can be derived from the basic widget using the inheritance mechanism. A widget maintains internal data such as the geometry of the window, back ground and fore ground colors of the window, cursor shape and size, etc. The methods supported by a widget manipulate the stored data and carry out operations such as resize window, iconify window, destroy window, etc.

Widgets are becoming the standard components of GUI design. This has given rise to the technique of component-based user interface development. We shall discuss more about widgets and component-based user interface development in Chapter 9 where we discuss GUI design.

7.1.6 Advantages and Disadvantages of OOD

As is true for any other technique, OOD has its own peculiar set of advantages and disadvantages. We briefly review these in the following.

Advantages of OOD

In the last couple of decades since OOD has come into existence, it has found widespread acceptance in industry as well as in academic circles. The main reason for the popularity of OOD is that it holds out the following promises:

- Code and design reuse
- Increased productivity
- Ease of testing and maintenance
- Better code and design understandability enabling development of large programs

Out of all the above mentioned advantages, it is usually agreed that the chief advantage of OOD is improved productivity—which comes about due to a variety of factors, such as the following:

- Code reuse by the use of predeveloped class libraries
- Code reuse due to inheritance
- Simpler and more intuitive abstraction, i.e., better management of inherent problem and code complexity
- Better problem decomposition

Several research results indicate that when companies start to develop software using the object-oriented paradigm, the first few projects incur higher costs than the traditionally developed projects. This is possibly due to getting used to a new technique and building up the class libraries that can be reused in the subsequent projects. After completion of a few projects, cost

saving becomes possible. According to experience reports, well-established object-oriented development environment can help to reduce development costs by as much as 20 per cent to 50 per cent over a traditional development environment.

Disadvantages of OOD

The following are some of the prominent disadvantages inherent to the object paradigm:

- The principles of abstraction, data hiding, inheritance, etc. do incur run time overhead due to the additional code that gets generated on account of these features. This causes an project-oriented program to run a little slower than an equivalent procedural program.
- An important consequence of object-orientation is that the data that is centralised in a procedural implementation, gets scattered across various objects in an object-oriented implementation. Therefore, the spatial locality of data becomes weak and this leads to higher cache miss ratios and consequently to larger memory access times. This finally shows up as increased program run time.

As we can see, increased run time is the principal disadvantage of object-orientation and higher productivity is the major advantage. In the present times, computers have become remarkably fast, and a small run time overhead is not an issue at all. Consequently, the advantages of OOD overshadow the disadvantages.

7.2 UNIFIED MODELLING LANGUAGE (UML)

As the name itself implies, UML is a language for documenting models. As is the case with any other language, UML has its syntax (a set of basic symbols and sentence formation rules) and semantics (meanings of basic symbols and sentences). It provides a set of basic graphical notations (e.g. rectangles, lines, ellipses, etc.) that can be combined in certain ways to document the design and analysis results.

It is important to remember that UML is neither a system design or development methodology by itself, nor is tied to any specific methodology. UML is merely a language for documenting models. Before the advent of UML, every design methodology not only prescribed entirely different design steps, but each was tied to some specific design modelling language. For example,

OMT methodology had its own design methodology and had its own unique set of notations. So was the case with Booch's methodology, and so on. This situation made it hard for someone familiar with one methodology to understand the design solutions developed and documented using another methodology. In general, reuse of design solutions across different methodologies was hard. UML was intended to address this problem that was inherent to the modelling techniques that existed.

UML can be used to document object-oriented analysis and design results that have been obtained using any methodology.

One of the objectives of the developers of UML was to keep the notations of UML independent of any specific design methodology, so that it can be used along with any specific design methodology. In this respect, UML is different from its predecessors (e.g., OMT, Booch's methodology, etc.) where the notations supported by the modelling languages were closely tied to the corresponding design methodologies.

7.2.1 Origin of UML

In the late eighties and early nineties, there was a proliferation of object-oriented design techniques and notations. Many of these had become extremely popular and were widely used. However, the notations they used and the specific design paradigms that they advocated, differed from each other in major ways. With so many popular techniques to choose from, it was not very uncommon to find different project teams in the same organisation using different methodologies and documenting their object-oriented analysis and design results using different notations. These diverse notations used for documenting design solutions gave arise to a lot of confusion among the team members and made it extremely difficult to reuse designs across projects and communicating ideas across project teams.

UML was developed to standardise the large number of object-oriented modelling notations that existed in the early nineties. The principal ones in use those days include the following:

- OMT [Rumbaugh 1991]
- Booch's methodology [Booch 1991]
- OOSE [Jacobson 1992]
- Odell's methodology [Odell 1992]

- Shlaer and Mellor methodology[Shlaer 1992]

Needless to say that UML has borrowed many concepts from these modeling techniques. Concepts and notations from especially the first three methodologies have heavily been drawn upon. The influence of various object modeling techniques on UML is shown schematically in Figure 7.12. As shown in Figure 7.12, OMT had the most profound influence on UML.

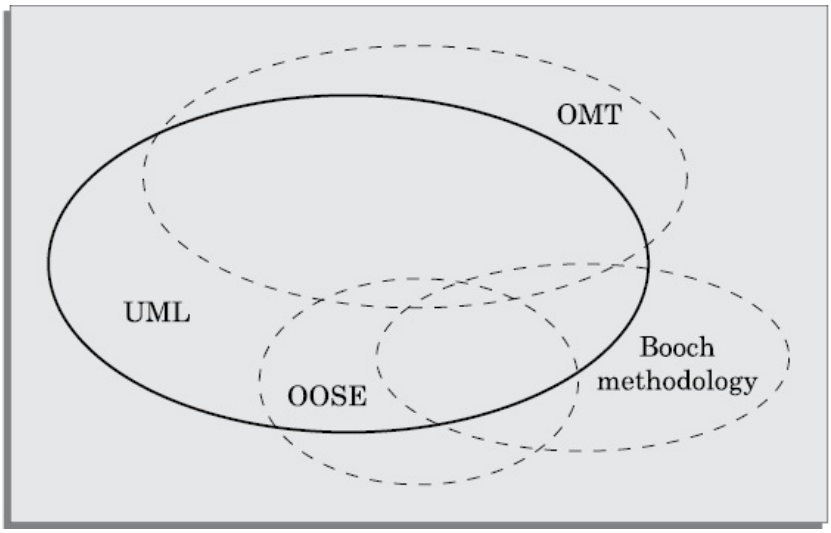

Figure 7.12: Schematic representation of the impact of different object modelling techniques on UML.

UML was adopted by object management group (OMG) as a de facto standard in 1997. Actually, OMG is not a standards formulating body, but is an association of industries that tries to facilitate early formulation of standards. OMG aims to promote consensus notations and techniques with the hope that if the usage becomes wide-spread, then they would automatically become standards. For more information on OMG, see www.omg.org. With widespread use of UML, ISO adopted UML a standard (ISO 19805) in 2005, and with this UML has become an official standard; this has further enhanced the use of UML.

UML is more complex than its antecedents. This is only natural and expected because it is intended to be more comprehensive and applicable to a wider gamut of problems than any of the notations that existed before UML. UML contains an extensive set of notations to help document several aspects (views) of a design solution through many types of diagrams. UML has successfully been used to model both large and small problems. The elegance of UML, its adoption by OMG, and subsequently by ISO as well as a strong industry backing have helped UML to find wide spread acceptance. UML is now being used in academic and research institutions as well as in large

number of software development projects world-wide. It is interesting to note that the use of UML is not restricted to the software industry alone. As an example of UML's use outside the software development problems, some car manufacturers are planning to use UML for their "build-to-order" initiative.

Many of the UML notations are difficult to draw by hand on a paper and are best drawn using a CASE tool such as Rational Rose© (see www.rational.com) or MagicDraw (www.magicdraw.com). Now several free UML CASE tools are also available on the web. Most of the available CASE tools help to refine an initial object model to final design, and these also automatically generate code templates in a variety of languages, once the UML models have been constructed.

7.2.2 Evolution of UML

Since the release of UML 1.0 in 1997, UML continues to evolve (see Figure 7.13) with feedback from practitioners and academicians to make it applicable to different system development situations. Almost every year several new releases (shown as UML 1.X in Figure 7.13) were announced. A major milestone in the evolution of UML was the release of UML 2.0 in the year 2007. Since the use of embedded applications is increasing rapidly, there was popular demand to extend UML to support the special concepts and notations required to develop embedded applications. UML 2.0 was an attempt to make UML applicable to the development of concurrent and embedded systems. For this, many new features such as events, ports, and frames in sequence diagrams were introduced. We briefly discuss these developments in this chapter.

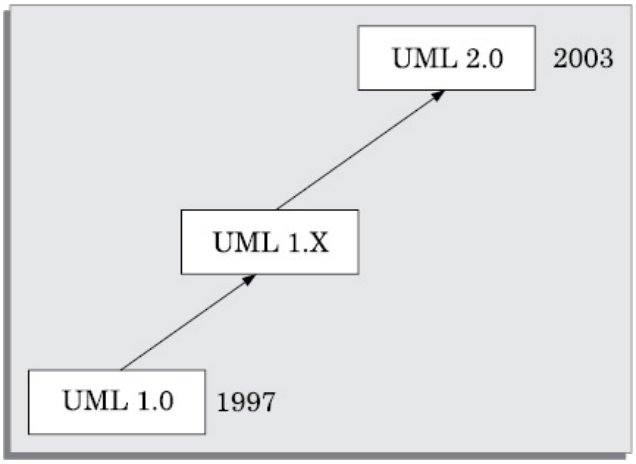

Figure 7.13: Evolution of UML.

What is a model?

Before we discuss the features of UML in detail, it is important to understand what exactly is meant by a model, and why is it necessary to create a model.

> A model is an abstraction of a real problem (or situation), and is constructed by leaving out unnecessary details. This reduces the problem complexity and makes it easy to understand the problem (or situation).

A model is a simplified version of a real system. It is useful to think of a model as capturing aspects important for some application while omitting (or abstracting out) the rest. As we had already pointed out in Chapter 1, as the size of a problem increases, the perceived complexity increases exponentially due to human cognitive limitations. Therefore, to develop a good understanding of any problem, it is necessary to construct a model of the problem. Modelling has turned out to be a very essential tool in software design and helps to effectively handle the complexity in a problem. These models that are first constructed are the models of the problem. A design methodology essentially transform these analysis models into a design model through iterative refinements.

Different types of models are obtained based on the specific aspects of the actual system that are ignored while constructing the model. To understand this, let us consider the models constructed by an architect of a large building. While constructing the frontal view of a large building (elevation plan), the architect ignores aspects such as floor plan, strength of the walls, details of the inside architecture, etc. While constructing the floor plan, he completely ignores the frontal view (elevation plan), site plan, thermal and lighting characteristics, etc. of the building.

A model in the context of software development can be graphical, textual, mathematical, or program code-based. Graphical models are very popular because they are easy to understand and construct. UML is primarily a graphical modeling tool. However, there are certain modelling situations (discussed later in this Chapter), for which in addition to the graphical UML models, separate textual explanations are required to accompany the graphical models.

Why construct a model?

An important reason behind constructing a model is that it helps to manage the complexity in a problem and facilitates arriving at good solutions and at the same time helps to reduce the design costs. The initial model of a problem is called an analysis model. The analysis model of a problem can be refined into a design model using a design

methodology. Once models of a system have been constructed, these can be used for a variety of purposes during software development, including the following:

- Analysis
- Specification
- Design
- Coding
- Visualisation and understanding of an implementation.
- Testing, etc.

Since a model can be used for a variety of purposes, it is reasonable to expect that the models would vary in detail depending on the purpose for which these are being constructed. For example, a model developed for initial analysis and specification should be very different from the one used for design. A model that is constructed for analysis and specification would not show any of the design decisions that would be made later on during the design stage. On the other hand, a model constructed for design purposes should capture all the design decisions. Therefore, it is a good idea to explicitly mention the purpose for which a model has been developed.

We now discuss the different types of UML diagrams and the notations used to develop these diagrams.

7.3 UML DIAGRAMS

In this section, we discuss the diagrams supported by UML 1.0. Later in Section 7.9.2, we discuss the changes to UML 1.0 brought about by UML 2.0. UML 1.0 can be used to construct nine different types of diagrams to capture five different views of a system. Just as a building can be modelled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc.; the different UML diagrams provide different perspectives of a software system to be developed and facilitate a comprehensive understanding of the system. Each perspective focuses on some specific aspect and ignores the rest. Some may ask, why construct several models from different perspectives—why not just construct one model that captures all perspectives? The answer to this is the following:

If a single model is made to capture all the required perspectives, then it would be as complex as the original problem, and would be of little use.

Once a system has been modelled from all the required perspectives, the constructed models can be refined to get the actual implementation of the system.

UML diagrams can capture the following views (models) of a system:

- User's view
- Structural view
- Behaviourial view
- Implementation view
- Environmental view

Figure 7.14 shows the different views that the UML diagrams can document. Observe that the users' view is shown as the central view. This is because based on the users' view, all other views are developed and all views need to conform to the user's view. Most of the object oriented analysis and design methodologies, including the one we are going to discuss in Chapter 8 require us to iterate among the different views several times to arrive at the final design. We first provide a brief overview of the different views of a system which can be documented using UML. In the subsequent sections, the diagrams used to realize the important views are discussed.

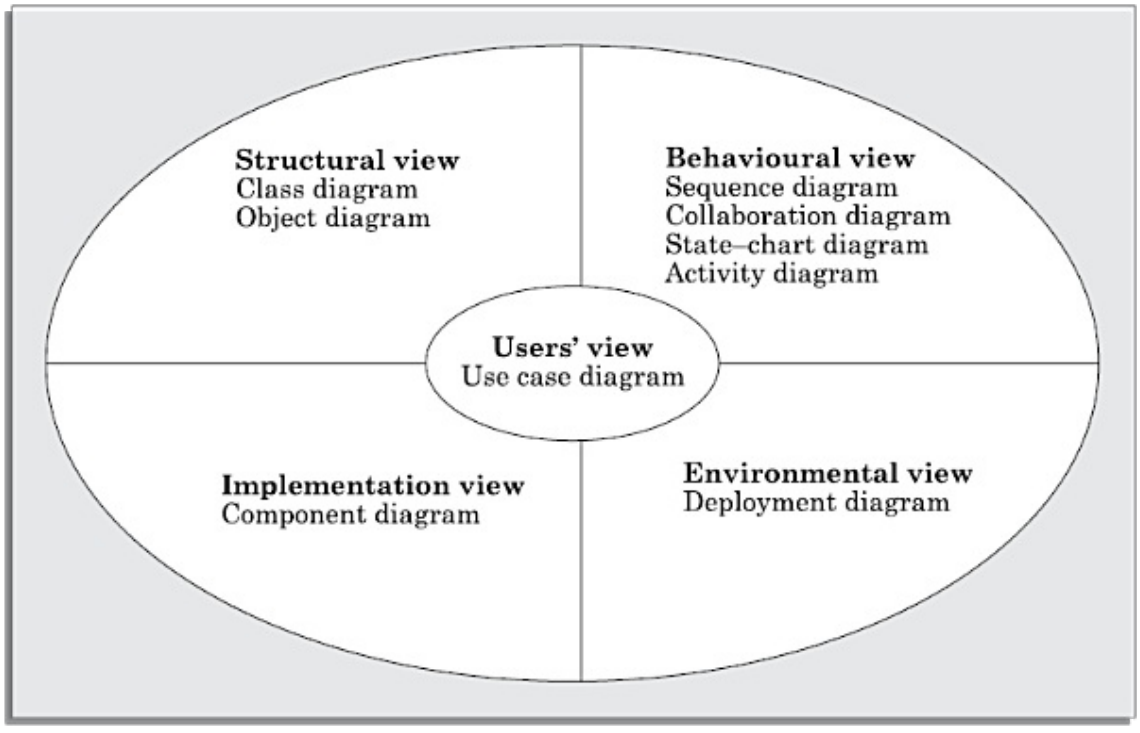

Figure 7.14: Different types of diagrams and views supported in UML.

Users' view

This view defines the functionalities made available by the system to its users.

> The users' view captures the view of the system in terms of the functionalities offered by the system to its users.

The users' view is a black-box view of the system where the internal structure, the dynamic behaviour of different system components, the implementation etc. are not captured. The users' view is very different from all other views in the sense that it is a functional model[1] compared to all other views that are essentially object models.[2]

The users' view can be considered as the central view and all other views are required to conform to this view. This thinking is in fact the crux of any user centric development style. It is indeed remarkable that even for object-oriented development, we need a functional view. That is because, after all, a user considers a system as providing a set of functionalities.

Structural view

The structural view defines the structure of the problem (or the solution) in terms of the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects).

> The structural model is also called the static model, since the structure of a system does not change with time.

Behaviourial view

The behaviourial view captures how objects interact with each other in time to realise the system behaviour. The system behaviour captures the time-dependent (dynamic) behaviour of the system. It therefore constitutes the dynamic model of the system.

Implementation view

This view captures the important components of the system and their interdependencies. For example, the implementation view might show the GUI part, the middleware, and the database part as the different parts and also would capture their interdependencies.

Environmental view

This view models how the different components are implemented on different pieces of hardware.

For any given problem, should one construct all the views using all the diagrams provided by UML? The answer is No. For a simple system, the use case model, class diagram, and one of the interaction diagrams may be sufficient. For a system in which the objects undergo many state changes, a state chart diagram may be necessary. For a system, which is implemented on a large number of hardware components, a deployment diagram may be necessary. So, the type of models to be constructed depends on the problem at hand. Rosenberg provides an analogy [Ros 2000] saying that "Just like you do not use all the words listed in the dictionary while writing a prose, you do not use all the UML diagrams and modeling elements while modeling a system."

7.4 USE CASE MODEL

The use case model for any system consists of a set of use cases.

Intuitively, the use cases represent the different ways in which a system can be used by the users.

A simple way to find all the use cases of a system is to ask the question —"What all can the different categories of users do by using the system?" Thus, for the library information system (LIS), the use cases could be:

- issue-book
- query-book
- return-book
- create-member
- add-book, etc.

Roughly speaking, the use cases correspond to the high-level functional requirements that we discussed in Chapter 4. We can also say that the use cases partition the system behaviour into transactions, such that each transaction performs some useful action from the user's point of view. Each transaction, to complete, may involve multiple message exchanges between the user and the system.

The purpose of a use case is to define a piece of coherent behaviour without revealing the internal structure of the system. The use cases do not mention any specific algorithm to be used nor the internal data representation, internal structure of the software. A use case typically

involves a sequence of interactions between the user and the system. Even for the same use case, there can be several different sequences of interactions. A use case consists of one main line sequence and several alternate sequences. The main line sequence represents the interactions between a user and the system that normally take place. The mainline sequence is the most frequently occurring sequence of interaction. For example, in the mainline sequence of the withdraw cash use case supported by a bank ATM would be—the user inserts the ATM card, enters password, selects the amount withdraw option, enters the amount to be withdrawn, completes the transaction, and collects the amount. Several variations to the main line sequence (called alternate sequences) may also exist. Typically, a variation from the mainline sequence occurs when some specific conditions hold. For the bank ATM example, consider the following variations or alternate sequences:

- Password is invalid.
- The amount to be withdrawn exceeds the account balance.

The mainline sequence and each of the alternate sequences corresponding to the invocation of a use case is called a scenario of the use case.

A use case can be viewed as a set of related scenarios tied together by a common goal. The main line sequence and each of the variations are called scenarios or instances of the use case. Each scenario is a single path of user events and system activity.

Normally, each use case is independent of the other use cases. However, implicit dependencies among use cases may exist because of dependencies that may exist among use cases at the implementation level due to factors such as shared resources, objects, or functions. For example, in the Library Automation System example, `renew-book` and `reserve-book` are two independent use cases. But, in actual implementation of renew-book, a check is to be made to see if any book has been reserved by a previous execution of the `reserve-book` use case. Another example of dependence among use cases is the following. In the Bookshop Automation Software, `update-inventory` and `sale-book` are two independent use cases. But, during execution of sale-book there is an implicit dependency on `update-inventory`. Since when sufficient quantity is unavailable in the inventory, `sale-book` cannot operate until the inventory is replenished using `update-inventory`.

The use case model is an important analysis and design artifact. As already

mentioned, other UML models must conform to this model in any use case-driven (also called as the user-centric) analysis and development approach. It should be remembered that the "use case model" is not really an object-oriented model according to a strict definition of the term.

> In contrast to all other types of UML diagrams, the use case model represents a functional or process model of a system.

7.4.1 Representation of Use Cases

A use case model can be documented by drawing a use case diagram and writing an accompanying text elaborating the drawing. In the use case diagram, each use case is represented by an ellipse with the name of the use case written inside the ellipse. All the ellipses (i.e. use cases) of a system are enclosed within a rectangle which represents the system boundary. The name of the system being modeled (e.g., library information system) appears inside the rectangle.

The different users of the system are represented by using stick person icons. Each stick person icon is referred to as an actor.[3] An actor is a role played by a user with respect to the system use. It is possible that the same user may play the role of multiple actors. An actor can participate in one or more use cases. The line connecting an actor and the use case is called the communication relationship. It indicates that an actor makes use of the functionality provided by the use case.

Both human users and external systems can be represented by stick person icons. When a stick person icon represents an external system, it is annotated by the stereotype <<external system>>.

At this point, it is necessary to explain the concept of a stereotype in UML. One of the main objectives of the creators of the UML was to restrict the number of primitive symbols in the language. It was clear to them that when a language has a large number of primitive symbols, it becomes very difficult to learn use. To convince yourself, consider that English with 26 alphabets is much easier to learn and use compared to the Chinese language that has thousands of symbols. In this context, the primary objective of stereotype is to reduce the number of different types of symbols that one needs to learn.

> The stereotype construct when used to annotate a basic symbol, can give slightly different meaning to the basic symbol— thereby eliminating the need to have several symbols whose meanings differ slightly from each other.

Just as you stereotype your friends as studious, jovial, serious, etc. stereotyping can be used to give special meaning to any basic UML construct. We shall, later on, see how other UML constructs can be stereotyped. We can stereotype the stick person icon symbol to denote an external system. If the developers of UML had assigned a separate symbol to denote things such as an external system, then the number of basic symbols one would have to learn and remember while using UML would have increased significantly. This would have certainly made learning and using UML much more difficult.

You can draw a rectangle around the use cases, called the system boundary box, to indicates the scope of your system. Anything within the box represents functionality that is in scope and anything outside the box is not. However, drawing the system boundary is optional.

We now give a few examples to illustrate how use cases of a system can be documented.

Example 7.2 The use case model for the Tic-tac-toe game software is shown in Figure 7.15. This software has only one use case, namely, "play move". Note that we did not name the use case "get-user-move", as "get-user-move" would be inappropriate because this would represent the developer's perspective of the use case. The use cases should be named from the users' perspective.

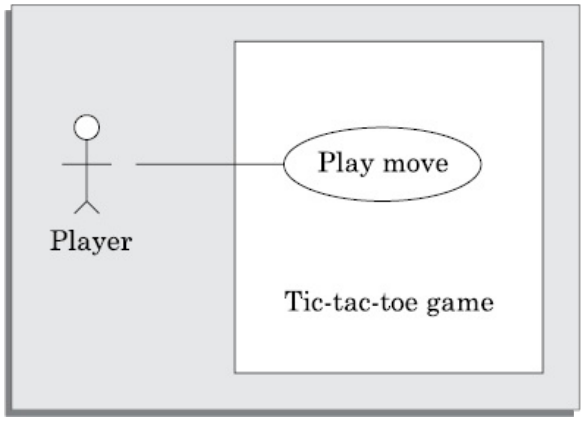

Figure 7.15: Use case model for Example 7.2.

Text description

Each ellipse in a use case diagram, by itself conveys very little information, other than giving a hazy idea about the use case. Therefore, every use case diagram should be accompanied by a text description. The text description should define the details of the interaction between the user and the computer as well as other relevant aspects of the use case. It should include all the behaviour

associated with the use case in terms of the mainline sequence, various alternate sequences, the system responses associated with the use case, the exceptional conditions that may occur in the behaviour, etc. The behaviour description is often written in a conversational style describing the interactions between the actor and the system. The text description may be informal, but some structuring is helpful. The following are some of the information which may be included in a use case text description in addition to the mainline sequence, and the alternate scenarios.

Contact persons: This section lists of personnel of the client organisation with whom the use case was discussed, date and time of the meeting, etc.

Actors: In addition to identifying the actors, some information about actors using a use case which may help the implementation of the use case may be recorded.

Pre-condition: The preconditions would describe the state of the system before the use case execution starts.

Post-condition: This captures the state of the system after the use case has successfully completed.

Non-functional requirements: This could contain the important constraints for the design and implementation, such as platform and environment conditions, qualitative statements, response time requirements, etc.

Exceptions, error situations: This contains only the domain-related errors such as lack of user's access rights, invalid entry in the input fields, etc. Obviously, errors that are not domain related, such as software errors, need not be discussed here.

Sample dialogs: These serve as examples illustrating the use case.

Specific user interface requirements: These contain specific requirements for the user interface of the use case. For example, it may contain forms to be used, screen shots, interaction style, etc.

Document references: This part contains references to specific domain-related documents which may be useful to understand the system operation.

Example 7.3 The use case diagram of the Super market prize scheme described in example 6.3 is shown in Figure 7.16.

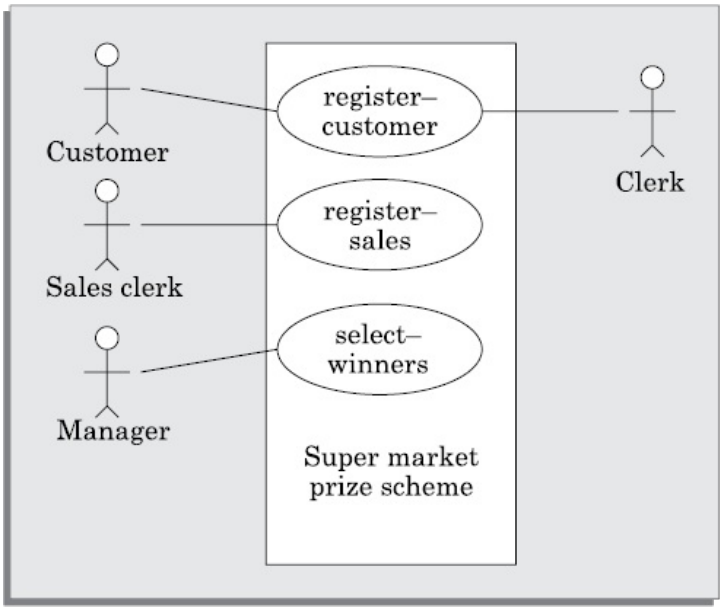

Figure 7.16: Use case model for Example 7.3.

Text description

U1: register-customer: Using this use case, the customer can register himself by providing the necessary details.

Scenario 1: Mainline sequence

1. Customer: select register customer option
2. System: display prompt to enter name, address, and telephone number.
3. Customer: enter the necessary values
4. System: display the generated id and the message that the customer has successfully been registered.

Scenario 2: At step 4 of mainline sequence

4 : System: displays the message that the customer has already registered.

Scenario 3: At step 4 of mainline sequence

4 : System: displays message that some input information have not been entered. The system displays a prompt to enter the missing values.

U2: register-sales: Using this use case, the clerk can register the details of the purchase made by a customer.

Scenario 1: Mainline sequence

1. Clerk: selects the register sales option.

2. System: displays prompt to enter the purchase details and the id of the customer.
3. Clerk: enters the required details.
4 : System: displays a message of having successfully registered the sale.

U3: select-winners. Using this use case, the manager can generate the winner list.

Scenario 2: Mainline sequence

1. Manager: selects the select-winner option.
2. System: displays the gold coin and the surprise gift winner list.

7.4.2 Why Develop the Use Case Diagram?

If you examine a use case diagram, the utility of the use cases represented by the ellipses would become obvious. They along with the accompanying text description serve as a type of requirements specification of the system and the model based on which all other models are developed. In other words, the use case model forms the core model to which all other models must conform. But, what about the actors (stick person icons)? What way are they useful to system development? One possible use of identifying the different types of users (actors) is in implementing a security mechanism through a login system, so that each actor can invoke only those functionalities to which he is entitled to. Another important use is in designing the user interface in the implementation of the use case targetted for each specific category of users who would use the use case. Another possible use is in preparing the documentation (e.g. users' manual) targeted at each category of user. Further, actors help in identifying the use cases and understanding the exact functioning of the system.

7.4.3 How to Identify the Use Cases of a System?

Identification of the use cases involves brain storming and reviewing the SRS document. Typically, the high-level requirements specified in the SRS document correspond to the use cases. In the absence of a well-formulated SRS document, a popular method of identifying the use cases is actor-based. This involves first identifying the different types of actors and their usage of the system. Subsequently, for each actor the

different functions that they might initiate or participate are identified. For example, for a Library Automation System, the categories of users can be members, librarian, and the accountant. Each user typically focuses on a set of functionalities. Foe example, the member typically concerns himself with book issue, return, and renewal aspects. The librarian concerns himself with creation and deletion of the member and book records. The accountant concerns itself with the amount collected from membership fees and the expenses aspects.

7.4.4 Essential Use Case versus Real Use Case

Essential use cases are created during early requirements elicitation. These are also early problem analysis artifacts. They are independent of the design decisions and tend to be correct over long periods of time.

Real use cases describe the functionality of the system in terms of its actual current design committed to specific input/output technologies. Therefore, the real use cases can be developed only after the design decisions have been made. Real use cases are a design artifact. However, sometimes organisations commit to development contracts that include the detailed user interface specifications. In such cases, there is no distinction between the essential use case and the real use case.

7.4.5 Factoring of Commonality among Use Cases

It is often desirable to factor use cases into component use cases. All use cases need not be factored. In fact, factoring of use cases are required under two situations as follows:

- Complex use cases need to be factored into simpler use cases. This would not only make the behaviour associated with the use case much more comprehensible, but also make the corresponding interaction diagrams more tractable. Without decomposition, the interaction diagrams for complex use cases may become too large to be accommodated on a single standard-sized (A4) paper.
- Use cases need to be factored whenever there is common behaviour across different use cases. Factoring would make it possible to define such behaviour only once and reuse it wherever required.

It is desirable to factor out common usage such as error handling from a set of use cases. This makes analysis of the class design much simpler and

elegant. However, a word of caution here. Factoring of use cases should not be done except for achieving the above two objectives. From the design point of view, it is not advantageous to break up a use case into many smaller parts just for the sake of it. UML offers three factoring mechanisms as discussed further.

Generalisation

Use case generalisation can be used when you have one use case that is similar to another, but does something slightly differently or something more. Generalisation works the same way with use cases as it does with classes. The child use case inherits the behaviour and meaning of the present use case. The notation is the same too (See Figure 7.17). It is important to remember that the base and the derived use cases are separate use cases and should have separate text descriptions.

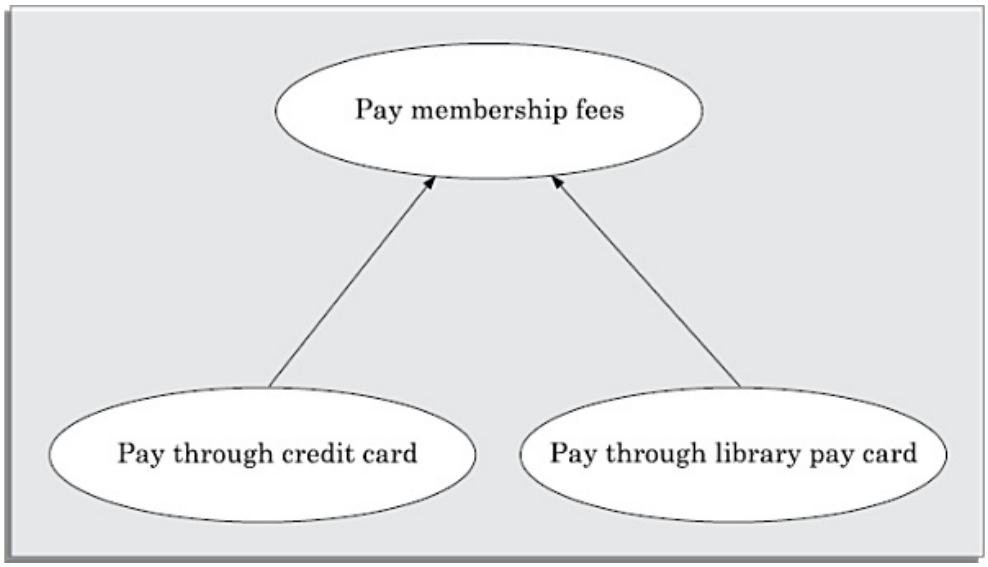

Figure 7.17: Representation of use case generalisation.

Includes

The includes relationship in the older versions of UML (prior to UML 1.1) was known as the uses relationship. The includes relationship implies one use case includes the behaviour of another use case in its sequence of events and actions. The includes relationship is appropriate when you have a chunk of behaviour that is similar across a number of use cases. The factoring of such behaviour will help in not repeating the specification and implementation across different use cases. Thus, the includes relationship explores the issue of reuse by factoring out the commonality across use cases. It can also be gainfully employed to

decompose a large and complex use case into more manageable parts.

As shown in Figure 7.18, the includes relationship is represented using a predefined stereotype <<include>>. In the includes relationship, a base use case compulsorily and automatically includes the behaviour of the common use case. As shown in example Figure 7.19, the use cases `issue-book` and `renew-book` both include `check-reservation` use case. The base use case may include several use cases. In such cases, it may interleave their associated common use cases together. The common use case becomes a separate use case and independent text description should be provided for it.

Figure 7.18: Representation of use case inclusion.

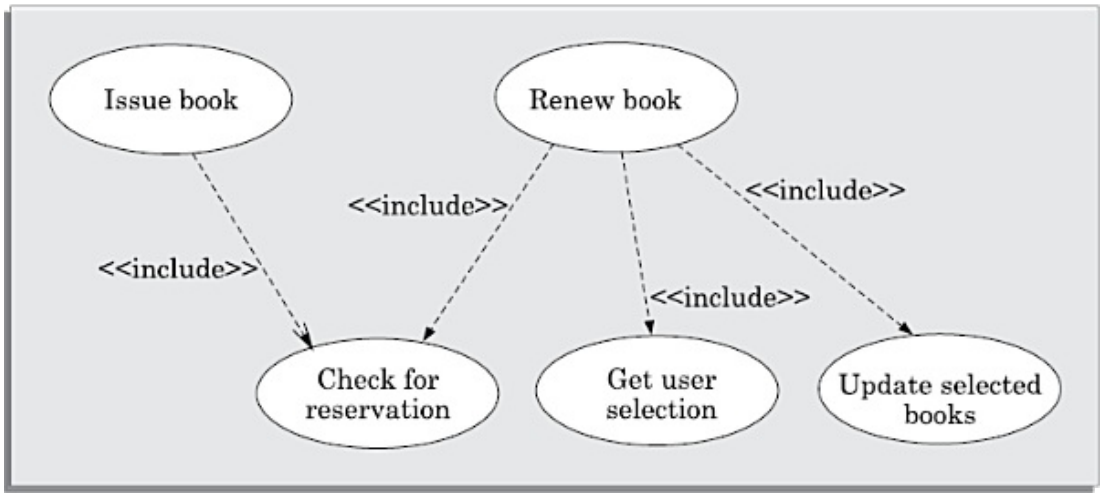

Figure 7.19: Example of use case inclusion.

Extends

The main idea behind the extends relationship among use cases is that it allows you show optional system behaviour. An optional system behaviour is executed only if certain conditions hold, otherwise the optional behaviour is not executed. This relationship among use cases is also predefined as a stereotype as shown in Figure 7.20.

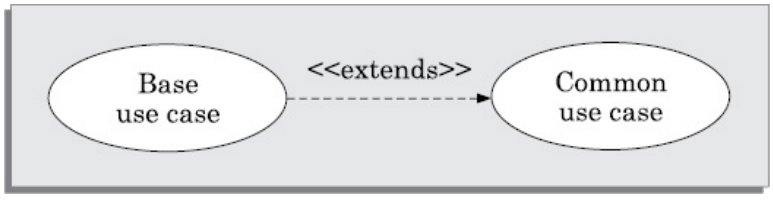

Figure 7.20: Example of use case extension.

The extends relationship is similar to generalisation. But unlike generalisation, the extending use case can add additional behaviour only at an extension point only when certain conditions are satisfied. The extension points are points within the use case where variation to the mainline (normal) action sequence may occur. The extends relationship is normally used to capture alternate paths or scenarios.

Organisation

When the use cases are factored, they are organised hierarchically. The high-level use cases are refined into a set of smaller and more refined use cases as shown in Figure 7.21. Top-level use cases are super-ordinate to the refined use cases. The refined use cases are sub-ordinate to the top-level use cases. Note that only the complex use cases should be decomposed and organised in a hierarchy. It is not necessary to decompose the simple use cases.

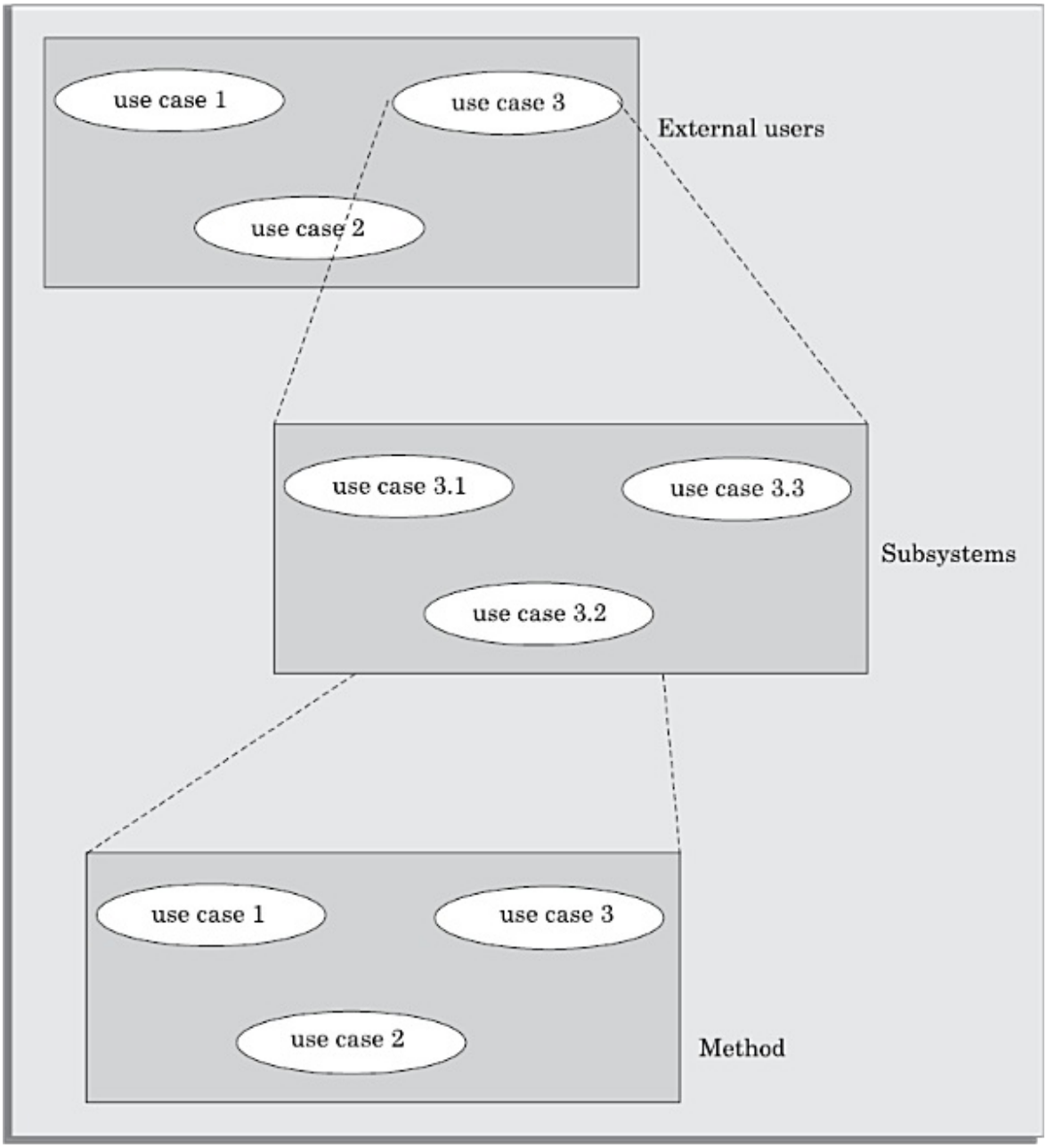

Figure 7.21: Hierarchical organisation of use cases.

The functionality of a super-ordinate use case is traceable to its subordinate use cases. Thus, the functionality provided by the super-ordinate use cases is composite of the functionality of the sub-ordinate use cases.

At the highest level of the use case model, only the fundamental use cases are shown. The focus is on the application context. Therefore, this level is also referred to as the context diagram. In the context diagram, the system limits are emphasised. In the top-level diagram, only those use cases with which external users interact are shown. The topmost use cases specify the complete services offered by the system to the external users of the system. The subsystem-level use cases specify the services offered by the

subsystems. Any number of levels involving the subsystems may be utilized. In the lowest level of the use case hierarchy, the class-level use cases specify the functional fragments or operations offered by the classes.

7.4.6 USE CASE PACKAGING

Packaging is the mechanism provided by UML to handle complexity. When we have too many use cases in the top-level diagram, we can package the related use cases so that at best 6 or 7 packages are present at the top level diagram. Any modeling element that becomes large and complex can be broken up into packages. Please note that you can put any element of UML (including another package) in a package diagram. The symbol for a package is a folder. Just as you organise a large collection of documents in a folder, you organise UML elements into packages. An example of packaging use cases is shown in Figure 7.22.

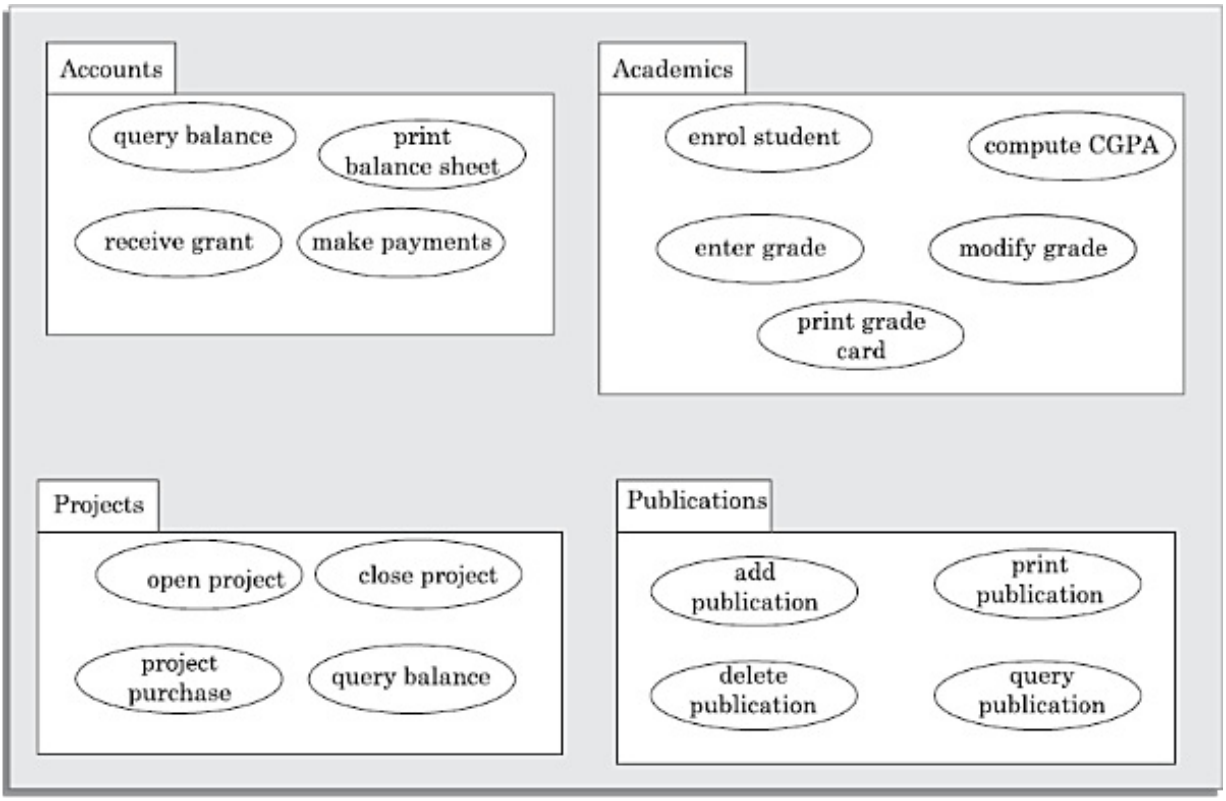

Figure 7.22: Use case packaging.

7.5 CLASS DIAGRAMS

A class diagram describes the static structure of a system. It shows how a system is structured rather than how it behaves. The static structure of a system comprises a number of class diagrams and their dependencies. The main constituents of a class diagram are classes and

their relationships—generalisation, aggregation, association, and various kinds of dependencies. We now discuss the UML syntax for representation of the classes and their relationships.

Classes

The classes represent entities with common features, i.e., attributes and operations. Classes are represented as solid outline rectangles with compartments. Classes have a mandatory name compartment where the name is written centered in boldface. The class name is usually written using mixed case convention and begins with an uppercase (e.g. `LibraryMember`). Object names on the other hand, are written using a mixed case convention, but starts with a small case letter (e.g., `studentMember`). Class names are usually chosen to be singular nouns. An example of various representations of a class are shown in Figure 7.23.

Classes have optional attributes and operations compartments. A class may appear on several diagrams. Its attributes and operations are suppressed on all but one diagram. But, one may wonder why there are so many representations for a class! The answer is that these different notations are used depending on the amount of information about a class is available. At the start of the design process, only the names of the classes is identified. This is the most abstract representation for the class. Later in the design process the methods for the class and the attributes are identified and the other more concrete notations are used.

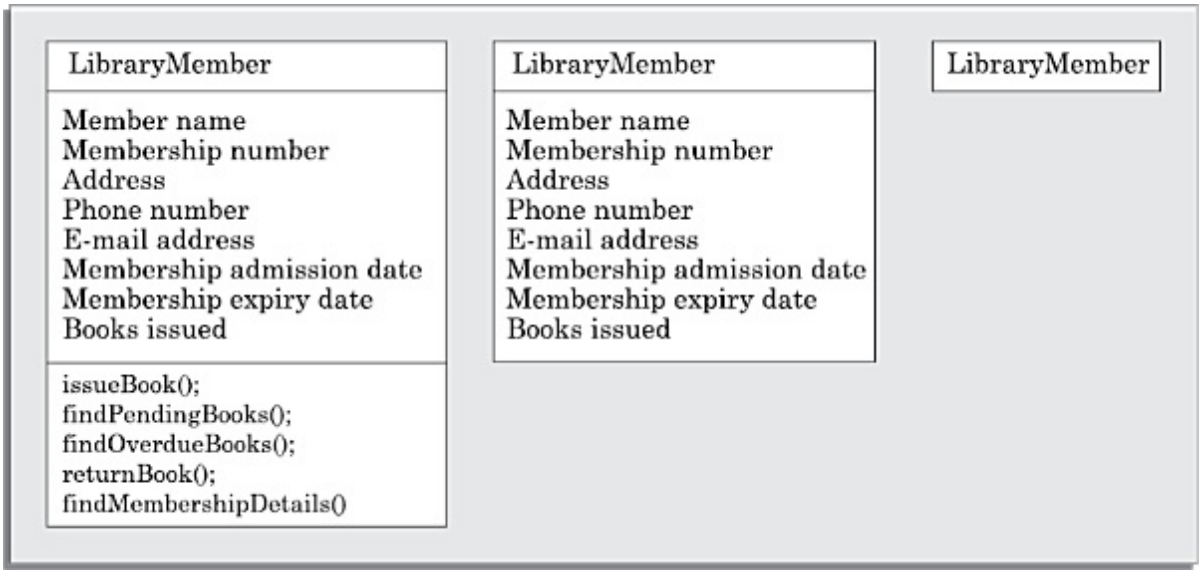

Figure 7.23: Different representations of the LibraryMember class.

Attributes

An attribute is a named property of a class. It represents the kind of data that an object might contain. Attributes are listed with their names, and may optionally contain specification of their type (that is, their class, e.g., Int, Book, Employee, etc.), an initial value, and constraints. Attribute names are written left-justified using plain type letters, and the names should begin with a lower case letter.

Attribute names may be followed by square brackets containing a multiplicity expression, e.g. sensorStatus[10]. The multiplicity expression indicates the number of attributes per instance of the class. An attribute without square brackets must hold exactly one value. The type of an attribute is written by following the attribute name with a colon and the type name, (e.g., sensorStatus[1]:Int).

The attribute name may be followed by an initialisation expression. The initialisation expression can consist of an equal sign and an initial value that is used to initialise the attributes of the newly created objects, e.g. sensorStatus[1]:Int=0.

Operation: The operation names are typically left justified, in plain type, and always begin with a lower case letter. Abstract operations are written in italics.[4] (Remember that abstract operations are those for which the implementation is not provided during the class definition.) The parameters of a function may have a kind specified. The kind may be "in" indicating that the parameter is passed into the operation; or "out" indicating that the parameter is only returned from the operation; or "inout" indicating that the parameter is used for passing data into the operation and getting result from the operation. The default is "in".

An operation may have a return type consisting of a single return type expression, e.g., issueBook(in bookName):Boolean. An operation may have a class scope (i.e., shared among all the objects of the class) and is denoted by underlining the operation name.

Often a distinction is made between the terms operation and method. An operation is something that is supported by a class and invoked by objects of other classes. There can be multiple methods implementing the same operation. We have pointed out earlier that this is called static polymorphism. The method names can be the same; however, it should be possible to distinguish among the methods by examining their parameters. Thus, the terms operation and method are distinguishable only when there is

polymorphism. When there is only a single method implementing an operation, the terms method and operation are indistinguishable and can be used interchangeably.

Association

Association between two classes is represented by drawing a straight line between the concerned classes. Figure 7.24 illustrates the graphical representation of the association relation. The name of the association is written along side the association line. An arrowhead may be placed on the association line to indicate the reading direction of the association. The arrowhead should not be misunderstood to be indicating the direction of a pointer implementing an association. On each side of the association relation, the multiplicity is noted as an individual number or as a value range. The multiplicity indicates how many instances of one class are associated with the other. Value ranges of multiplicity are noted by specifying the minimum and maximum value, separated by two dots, e.g. 1..5. An asterisk is used as a wild card and means many (zero or more). The association of Figure 7.24 should be read as "Many books may be borrowed by a LibraryMember". Usually, associations (and links) appear as verbs in the problem statement.

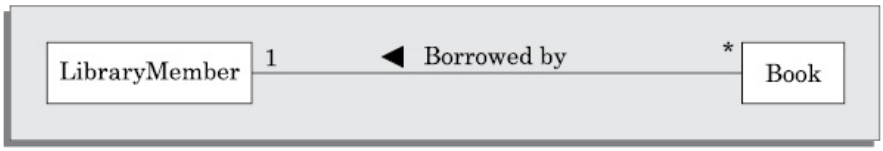

Figure 7.24: Association between two classes.

Associations are usually realised by assigning appropriate reference attributes to the classes involved. Thus, associations can be implemented using pointers from one object class to another. Links and associations can also be implemented by using a separate class that stores which objects of a class are linked to which objects of another class. Some CASE tools use the role names of the association relation for the corresponding automatically generated attribute.

Aggregation

Aggregation is a special type of association relation where the involved classes are not only associated to each other, but a whole-part relationship exists between them. That is, the aggregate object not only knows the addresses of its parts and therefore invoke the methods of its parts, but also takes the responsibility of creating and destroying

its parts. An example of aggregation, a book register is an aggregation of book objects. Books can be added to the register and deleted as and when required.

Aggregation is represented by an empty diamond symbol at the aggregate end of a relationship. An example of the aggregation relationship has been shown in Fig 7.25. The figure represents the fact that a document can be considered as an aggregation of paragraphs. Each paragraph can in turn be considered as aggregation of lines. Observe that the number 1 is annotated at the diamond end, and a * is annotated at the other end. This means that one document can have many paragraphs. On the other hand, if we wanted to indicate that a document consists of exactly 10 paragraphs, then we would have written number 10 in place of the (*).

The aggregation relationship cannot be reflexive (i.e. recursive). That is, an object cannot contain objects of the same class as itself. Also, the aggregation relation is not symmetric. That is, two classes A and B cannot contain instances of each other. However, the aggregation relationship can be transitive. In this case, aggregation may consist of an arbitrary number of levels. As an example of a transitive aggregation relationship, please see Figure 7.25.

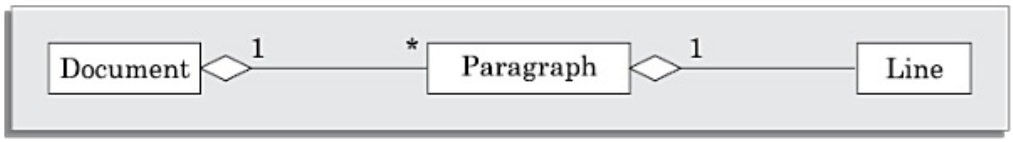

Figure 7.25: Representation of aggregation.

Composition

Composition is a stricter form of aggregation, in which the parts are existence-dependent on the whole. This means that the life of the parts cannot exist outside the whole. In other words, the lifeline of the whole and the part are identical. When the whole is created, the parts are created and when the whole is destroyed, the parts are destroyed.

A typical example of composition is an order object where after placing the order, no item in the order cannot be changed. If any changes to any of the order items are required after the order has been placed, then the entire order has to be cancelled and a new order has to be placed with the changed items. In this case, as soon as an order object is created, all the order items in it are created and as soon as the order object is destroyed, all order items in it are also destroyed. That is, the life of the components (order items) is the same as the aggregate (order). The composition relationship is

represented as a filled diamond drawn at the composite-end. An example of the composition relationship is shown in Figure 7.26.

Figure 7.26: Representation of composition.

Aggregation versus Composition: Both aggregation and composition represent part/whole relationships. When the components can dynamically be added and removed from the aggregate, then the relationship is aggregation. If the components cannot be dynamically added/delete then the components are have the same life time as the composite. In this case, the relationship is represented by composition.

As an example, consider the example of an order consisting many order items. If the order once placed, the items cannot be changed at all. In this case, the order is a composition of order items. However, if order items can be changed (added, delete, and modified) after the order has been placed, then aggregation relation can be used to model it.

Inheritance

The inheritance relationship is represented by means of an empty arrow pointing from the subclass to the superclass. The arrow may be directly drawn from the subclass to the superclass. Alternatively, when there are many subclasses of a base class, the inheritance arrow from the subclasses may be combined to a single line (see Figure 7.27) and is labelled with the aspect of the class that is abstracted.

The direct arrows allow flexibility in laying out the diagram and can easily be drawn by hand. The combined arrows emphasise the collectivity of the subclasses, when specialisation has been done on the basis of some discriminator. In the example of Figure 7.27, issuable and reference are the discriminators. The various subclasses of a superclass can then be differentiated by means of the discriminator. The set of subclasses of a class having the same discriminator is called a partition. It is often helpful to mention the discriminator during modelling, as these become documented design decisions.

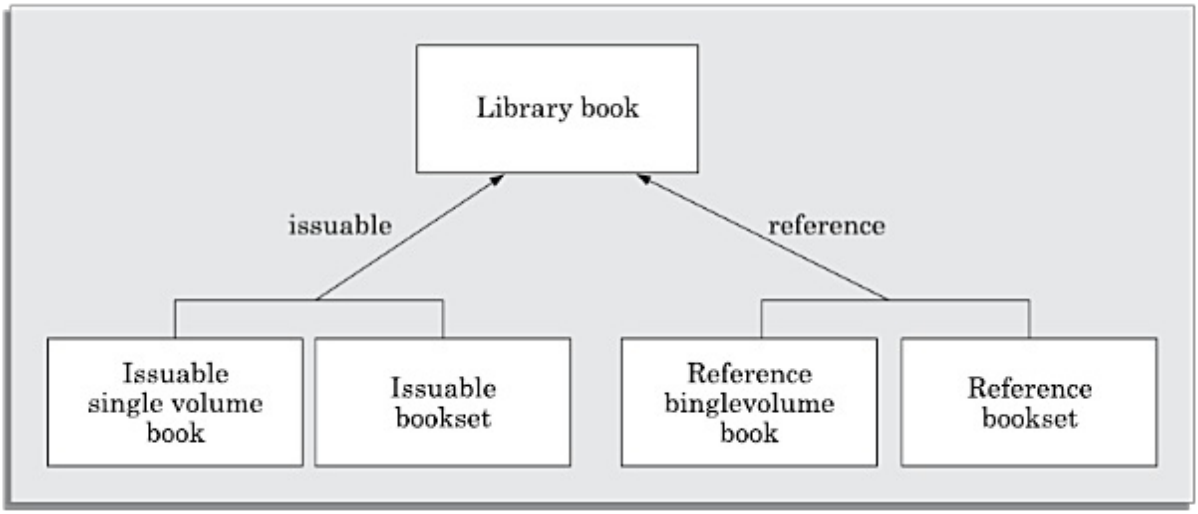

Figure 7.27: Representation of the inheritance relationship.

Dependency

A dependency relationship is shown as a dotted arrow (see Figure 7.28) that is drawn from the dependent class to the independent class.

Figure 7.28: Representation of dependence between classes.

Constraints

A constraint describes a condition or an integrity rule. Constraints are typically used to describe the permissible set of values of an attribute, to specify the pre- and post-conditions for operations, to define certain ordering of items, etc. For example, to denote that the books in a library are sorted on ISBN number we can annotate the book class with the constraint

{sorted}. UML allows you to use any free form expression to describe constraints. The only rule is that they are to be enclosed within braces. Constraints can be expressed using informal English. However, UML also provides object constraint language (OCL) to specify constraints. In OCL the constraints are specified a semi-formal language, and therefore it is more amenable to automatic processing as compared to the informal constraints enclosed within {}. The interested reader is referred to [Rumbaugh1999].

Object diagrams

Object diagrams shows the snapshot of the objects in a system at a point in

time. Since it shows instances of classes, rather than the classes themselves, it is often called as an instance diagram. The objects are drawn using rounded rectangles (see Figure 7.29).

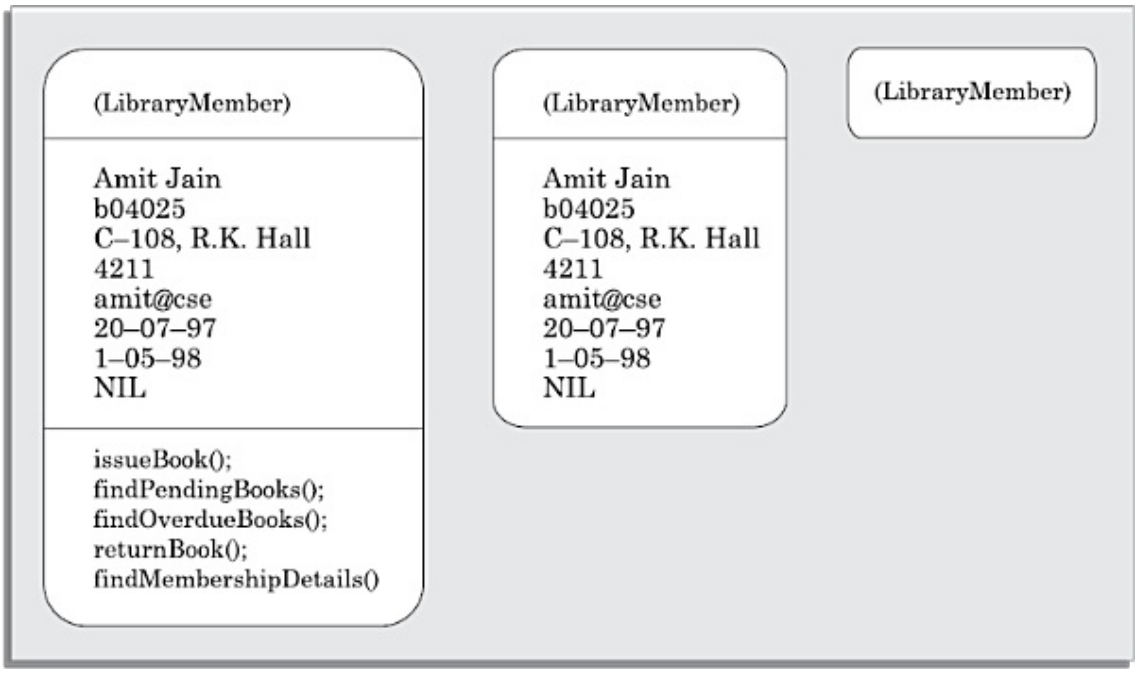

Figure 7.29: Different representations of a LibraryMember object.

An object diagram may undergo continuous change as execution proceeds. For example, links may get formed between objects and get broken. Objects may get created and destroyed, and so on. Object diagrams are useful to explain the working of a system.

7.6 INTERACTION DIAGRAMS

When a user invokes one of the functions supported by a system, the required behaviour is realised through the interaction of several objects in the system. Interaction diagrams, as their name itself implies, are models that describe how groups of objects interact among themselves through message passing to realise some behaviour.

Typically, each interaction diagram realises the behaviour of a single use case.

Sometimes, especially for complex use cases, more than one interaction diagrams may be necessary to capture the behaviour. An interaction diagram shows a number of example objects and the messages that are passed between the objects within the use case.

There are two kinds of interaction diagrams—sequence diagrams and collaboration diagrams. These two diagrams are equivalent in the sense that

any one diagram can be derived automatically from the other. However, they are both useful. These two actually portray different perspectives of behaviour of a system and different types of inferences can be drawn from them. The interaction diagrams play a major role in any effective object-oriented design process. We discuss this issue in Chapter 8.

Sequence diagram

A sequence diagram shows interaction among objects as a two dimensional chart. The chart is read from top to bottom. The objects participating in the interaction are shown at the top of the chart as boxes attached to a vertical dashed line. Inside the box the name of the object is written with a colon separating it from the name of the class and both the name of the object and the class are underlined. This signifies that we are referring any arbitrary instance of the class. For example, in Figure 7.30 :Book represents any arbitrary instance of the Book class.

An object appearing at the top of the sequence diagram signifies that the object existed even before the time the use case execution was initiated. However, if some object is created during the execution of the use case and participates in the interaction (e.g., a method call), then the object should be shown at the appropriate place on the diagram where it is created.

The vertical dashed line is called the object's lifeline. Any point on the lifeline implies that the object exists at that point. Absence of lifeline after some point indicates that the object ceases to exist after that point in time, particular point of time. Normally, at the point if an object is destroyed, the lifeline of the object is crossed at that point and the lifeline for the object is not drawn beyond that point. A rectangle called the activation symbol is drawn on the lifeline of an object to indicate the points of time at which the object is active. Thus an activation symbol indicates that an object is active as long as the symbol (rectangle) exists on the lifeline. Each message is indicated as an arrow between the lifelines of two objects. The messages are shown in chronological order from the top to the bottom. That is, reading the diagram from the top to the bottom would show the sequence in which the messages occur.

Each message is labelled with the message name. Some control information can also be included. Two important types of control information are:

- A condition (e.g., [invalid]) indicates that a message is sent, only if the condition is true.
- An iteration marker shows that the message is sent many times to multiple receiver objects as would happen when you are iterating over a collection or the elements of an array. You can also indicate the basis of the iteration, e.g., [for every book object].

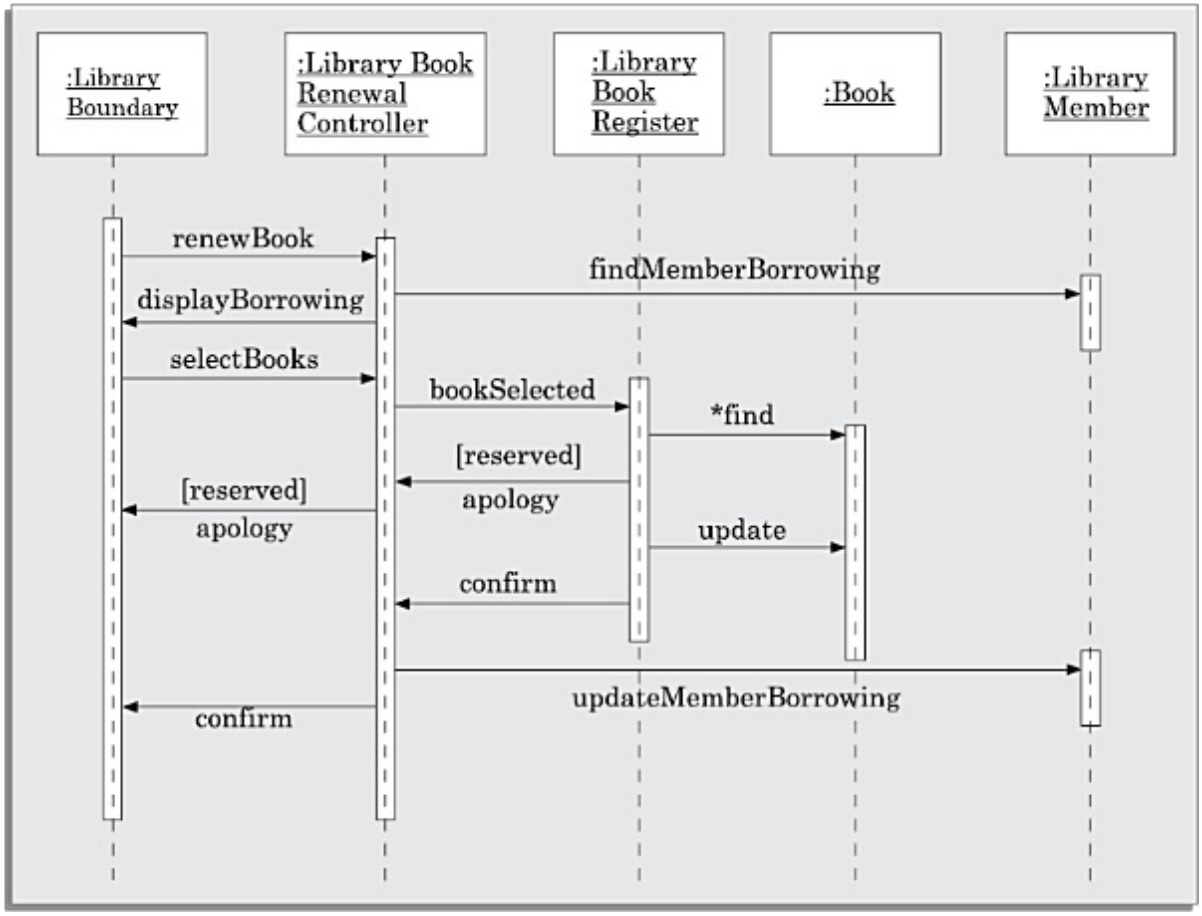

Figure 7.30: Sequence diagram for the renew book use case

The sequence diagram for the book renewal use case for the Library Automation Software is shown in Figure 7.30. Observe that the exact objects which participate to realise the renew book behaviour and the order in which they interact can be clearly inferred from the sequence diagram. The development of the sequence diagram in the development methodology (discussed in Chapter 8) would help us to determine the responsibilities that must be assigned to the different classes; i.e., what methods should be supported by each class.

Collaboration diagram

A collaboration diagram shows both structural and behavioural aspects

explicitly. This is unlike a sequence diagram which shows only the behavioural aspects. The structural aspect of a collaboration diagram consists of objects and links among them indicating association. In this diagram, each object is also called a collaborator. The behavioural aspect is described by the set of messages exchanged among the different collaborators.

The link between objects is shown as a solid line and can be used to send messages between two objects. The message is shown as a labelled arrow placed near the link. Messages are prefixed with sequence numbers because they are the only way to describe the relative sequencing of the messages in this diagram.

The collaboration diagram for the example of Figure 7.30 is shown in Figure 7.31. Use of the collaboration diagrams in our development process would be to help us to determine which classes are associated with which other classes.

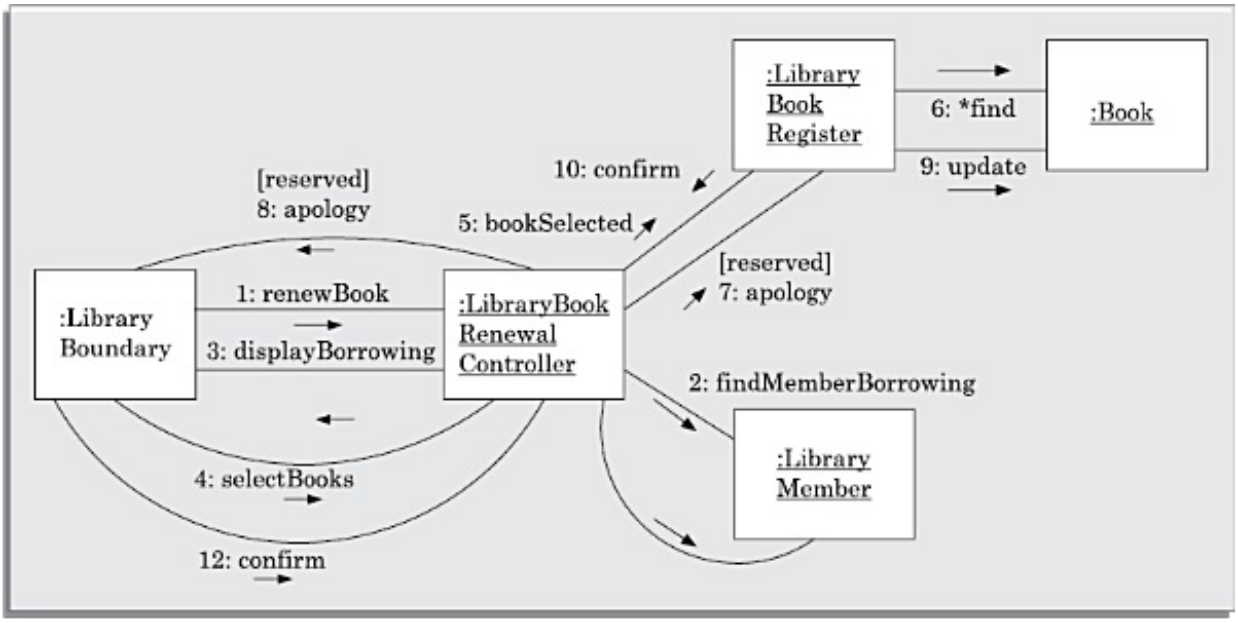

Figure 7.31: Collaboration diagram for the renew book use case.

7.7 ACTIVITY DIAGRAM

The activity diagram is possibly one modelling element which was not present in any of the predecessors of UML. No such diagrams were present either in the works of Booch, Jacobson, or Rumbaugh. It has possibly been based on the event diagram of Odell [1992] though the notation is very different from that used by Odell.

The activity diagram focuses on representing various activities or chunks of processing and their sequence of activation. The activities in general may not

correspond to the methods of classes. An activity is a state with an internal action and one or more outgoing transitions which automatically follow the termination of the internal activity. If an activity has more than one outgoing transitions, then exact situation under which each is executed must be identified through appropriate conditions.

Activity diagrams are similar to the procedural flow charts. The main difference is that activity diagrams support description of parallel activities and synchronisation aspects involved in different activities.

Parallel activities are represented on an activity diagram by using swim lanes. Swim lanes enable you to group activities based on who is performing them, e.g., academic department vs. hostel office. Thus swim lanes subdivide activities based on the responsibilities of some components. The activities in a swim lanes can be assigned to some model elements, e.g. classes or some component, etc. For example, in Figure 7.32 the swim lane corresponding to the academic section, the activities that are carried out by the academic section and the specific situation in which these are carried out are shown.

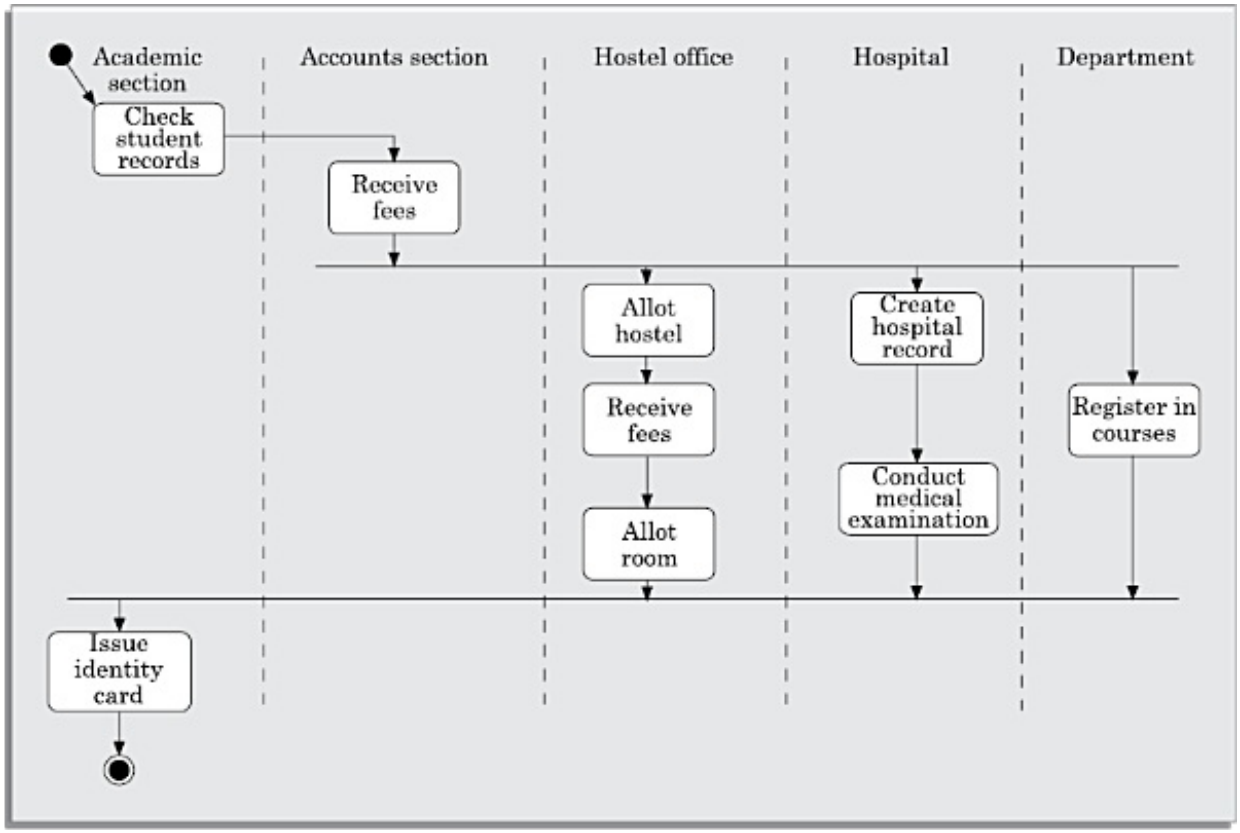

Figure 7.32: Activity diagram for student admission procedure at IIT.

Activity diagrams are normally employed in business process modelling. This is carried out during the initial stages of requirements analysis and specification. Activity diagrams can be very useful to understand complex

processing activities involving the roles played by many components. Besides helping the developer to understand the complex processing activities, these diagrams can also be used to develop interaction diagrams which help to allocate activities (responsibilities) to classes.

The student admission process in IIT is shown as an activity diagram in Figure 7.32. This shows the part played by different components of the Institute in the admission procedure. After the fees are received at the account section, parallel activities start at the hostel office, hospital, and the Department. After all these activities complete (this is a synchronisation issue and is represented as a horizontal line), the identity card can be issued to a student by the Academic section.

7.8 STATE CHART DIAGRAM

A state chart diagram is normally used to model how the state of an object changes in its life time. State chart diagrams are good at describing how the behaviour of an object changes across several use case executions. However, if we are interested in modelling some behaviour that involves several objects collaborating with each other, state chart diagram is not appropriate. We have already seen that such behaviour is better modelled using sequence or collaboration diagrams. State chart diagrams are based on the finite state machine (FSM) formalism. An FSM consists of a finite number of states corresponding to those of the object being modelled. The object undergoes state changes when specific events occur. The FSM formalism existed long before the object-oriented technology and has been used for a wide variety of applications. Apart from modelling, it has even been used in theoretical computer science as a generator for regular languages.

Why state chart?

A major disadvantage of the FSM formalism is the state explosion problem. The number of states becomes too many and the model too complex when used to model practical systems. This problem is overcome in UML by using state charts. The state chart formalism was proposed by David Harel [1990]. A state chart is a hierarchical model of a system and introduces the concept of a composite state (also called nested state).

Actions are associated with transitions and are considered to be processes that occur quickly and are not interruptible. Activities are associated with

states and can take longer. An activity can be interrupted by an event.

Basic elements of a state chart

The basic elements of the state chart diagram are as follows:

Initial state: This represented as a filled circle.

Final state: This is represented by a filled circle inside a larger circle.

State: These are represented by rectangles with rounded corners.

Transition: A transition is shown as an arrow between two states. Normally, the name of the event which causes the transition is places along side the arrow. You can also assign a guard to the transition. A guard is a Boolean logic condition. The transition can take place only if the guard evaluates to true. The syntax for the label of the transition is shown in 3 parts—[guard]event/action.

An example state chart for the order object of the Trade House Automation software is shown in Figure 7.33. Observe that from Rejected order state, there is an automatic and implicit transition to the end state. Such transitions are called pseudo transitions.

7.9 POSTSCRIPT

UML has gained rapid acceptance among practitioners and academicians over a short time and has proved its utility in arriving at good design solutions to software development problems.

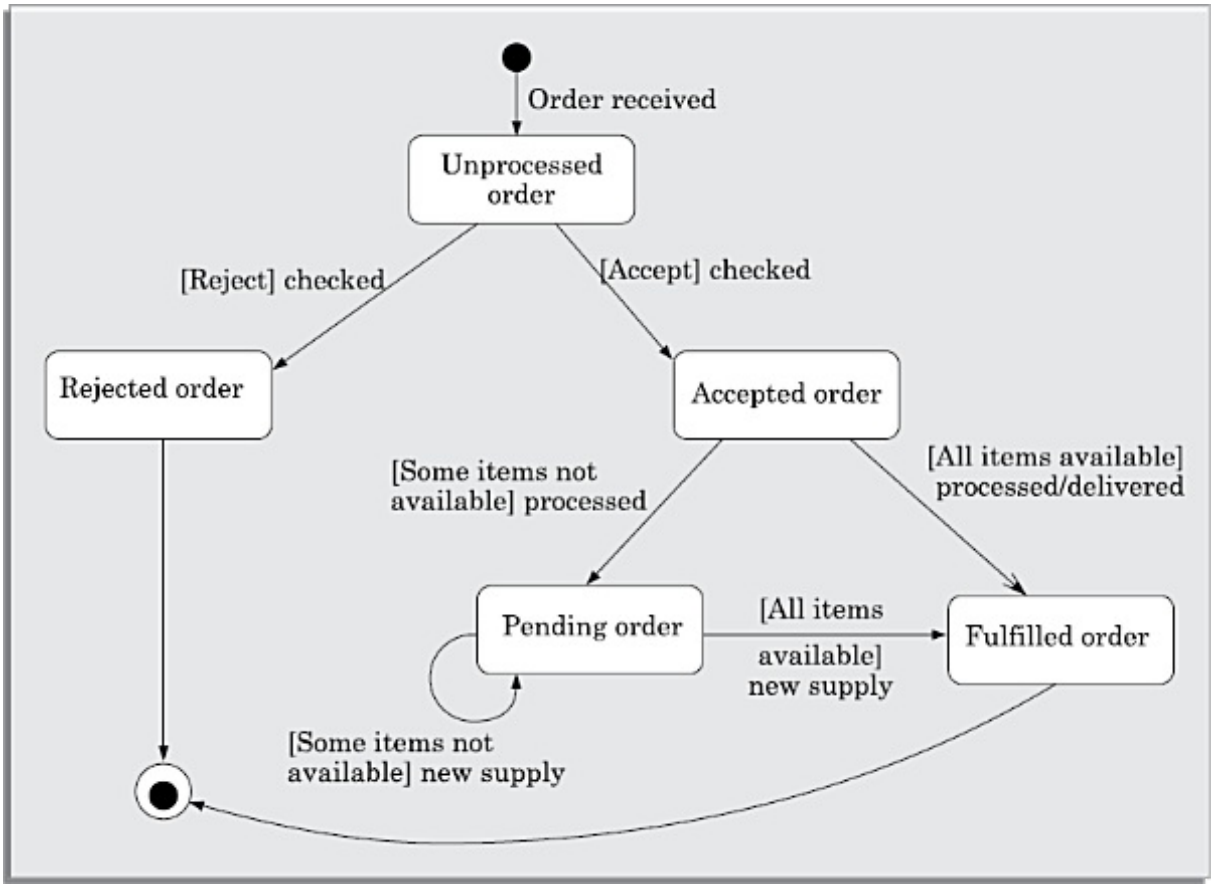

Figure 7.33: State chart diagram for an order object.

In this text, we have kept our discussions on UML to a bare minimum and have concentrated only on those aspects that are necessary to solve moderate sized traditional software design problems.

Before concluding this chapter, we give an overview of some of the aspects that we had chosen to leave out. We first discuss the package and deployment diagrams. Since UML has undergone a significant change with the release of UML 2.0 in 2003. We briefly mention the highlights of the improvements brought about UML 2.0 over the UML 1.X which was our focus so far. This significant revision was necessitated to make UML applicable to the development of software for emerging embedded and telecommunication domains.

7.9.1 Package, Component, and Deployment Diagrams

In the following subsections we provide a brief overview of the package, component, and deployment diagrams:

Package diagram

A package is a grouping of several classes. In fact, a package diagram can be

used to group any UML artifacts. We had already discussed packaging of use cases in Section 7.4.6. Packages are popular way of organising source code files. Java packages are a good example which can be modelled using a package diagram. Such package diagrams show the different class groups (packages) and their inter dependencies. These are very useful to document organisation of source files for large projects that have a large number of program files. An example of a package diagram has been shown in Figure 7.34.

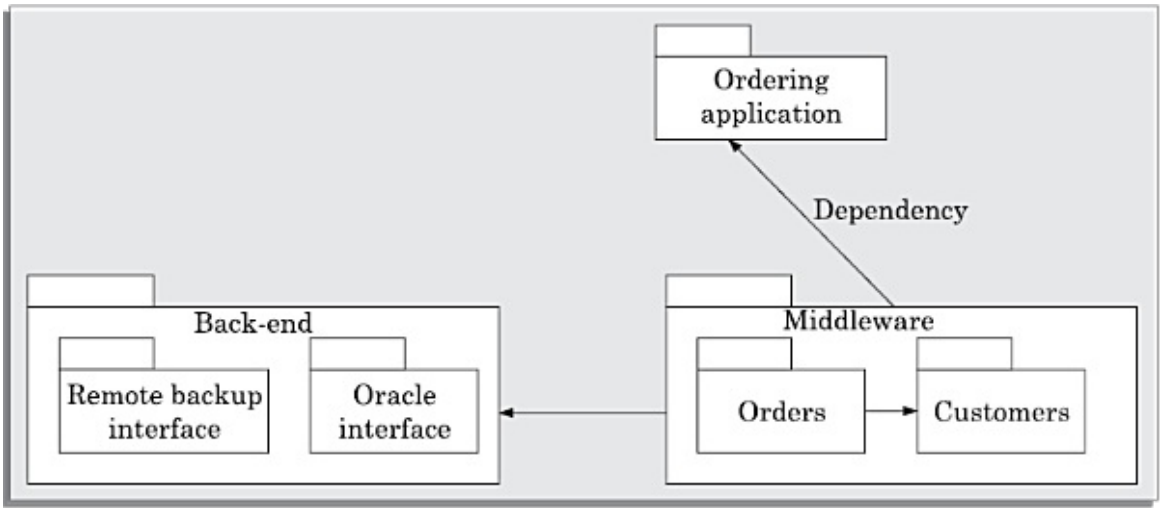

Figure 7.34: An example package diagram.

Note, that a package may contain further packages.

Component diagram

A component represents a piece of software that can be independently purchased, upgraded, and integrated into an existing software. A component diagram can be used to represent the physical structure of an implementation in terms of the various components of the system. A component diagram is typically used to achieve the following purposes:

- Organise source code to be able to construct executable releases.
- Specify dependencies among different components.

A package diagram can be used to provide a high-level view of each component in terms the different classes it contains.

Deployment diagram

The deployment diagram shows the environmental view of a system. That is, it captures the environment in which the software solution is implemented. In other words, a deployment diagram shows how a

software system will be physically deployed in the hardware environment. That is, which component will execute on which hardware component and how they will they communicate with each other. Since the diagram models the run time architecture of an application, this diagram can be very useful to the system's operation staff.

The environmental view provided by the deployment diagram is important for complex and large software solutions that run on hardware systems comprising multiple components. In this case, deployment diagram provides an overview of how the different components are distributed among the different hardware components of the system.

7.9.2 UML 2.0

UML 1.X lacked a few specialised capabilities that made it difficult to use in some non- traditional domains. Some of the features that prominently lacked in UML 1.X include lack of support for representation of the following—concurrent execution of methods, development domain, asynchronous messages, events, ports, and active objects. In many applications, including the embedded and telecommunication software development, capability to model timing requirements using a timing diagram was urgently required to make UML applicable in these important segments of software development. Further, certain changes were required to support interoperability among UML-based CASE tools using XML metadata interchange (XMI).

UML 2.0 defines thirteen types of diagrams, divided into three categories as follows:

Structure diagrams: These include the class diagram, object diagram, component diagram, composite structure diagram, package diagram, and deployment diagram.

Behaviour diagrams: These diagrams include the use case diagram, activity diagram, and state machine diagram.

Interaction diagrams: These diagrams include the sequence diagram, communication diagram, timing diagram, and interaction overview diagram. The collaboration diagram of UML 1.X has been renamed in UML 2.0 as communication diagram. This renaming was necessary as the earlier name was somewhat misleading, it shows the communications among the classes during the execution of a use case rather than showing collaborative problem solving.

Though a large number of new features have been introduced in UML 2.0 as compared to 1.X, in the following subsections, we discuss only two of the enhancements in UML2.0 through combined fragments and composite structure diagram.

Combined fragments in sequence diagrams

A combined fragment is a construct that has been introduced in UML 2.0 to allow description of various control and logic structures in a more visually apparent and concise manner. It also allows representation of concurrent execution behaviour such as that takes place in a mutithreaded execution situation.

Let us now understand the anatomy of a combined fragment and its use. A combined fragment divides a sequence diagram into a number of areas or fragments that have different behaviour (see Figure 7.35). A combined fragment appears over an area of a sequence diagram to make certain control and logic aspects visually clear. As shown in Figure 7.35, a combined fragment consists of many fragments and an operator shown at the top left corner. Each fragment can be associated with a guard (a Boolean expression). We now discuss these components of a combined fragment:

Fragment: A fragment in a sequence diagram is represented by a box, and encloses a portion of the interactions within a sequence diagram. Each fragment is also known as an interaction operand. An interaction operand may contain an optional guard condition, which is also called an interaction constraint. The behaviour specified in an interaction operand is executed only if its guard condition evaluates to true.

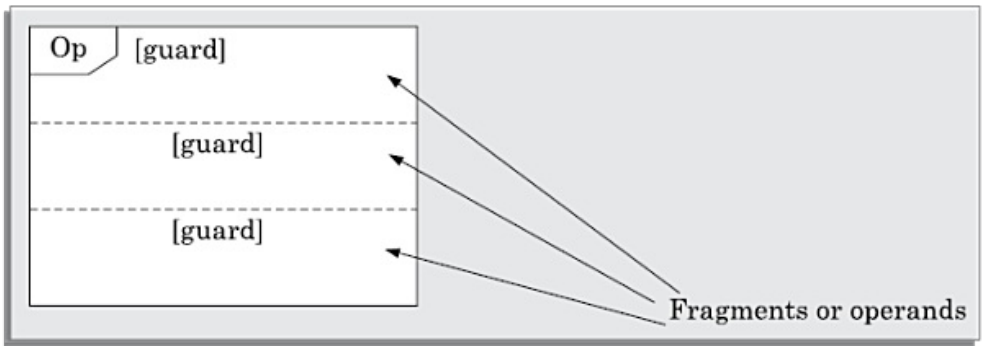

Figure 7.35: Anatomy of a combined fragment in UML 2.0.

Operator: A combined fragment is associated with one operator called interaction operator that is shown at the top left corner of the fragment. The operator indicates the type of fragment. The type of logic operator along with the guards in the fragment defines the behaviour of the

combined fragment. A combined fragment can also contain nested combined fragments or interaction uses containing additional conditional structures that represent more complex structures that affect the flow of messages.

Some of the important operators of a combined fragment are the following:

alt: This operator indicates that among multiple fragments, only the one whose guard is true will execute.

opt: An optional fragment that will execute only if the guard is true.

par: This operator indicated that various fragments can execute at the same time.

loop: A loop operator indicates that the various fragments may execute multiple times and the guard indicates the basis of iteration, meaning that the execution would continue until the guard turns false.

region: It defines a critical region in which only one thread can execute.

An example of a combined fragment has been shown in Figure 7.36.

Composite structure diagram

The composite structure diagram lets you define how a class is defined by a further structure of classes and the communication paths between these parts. Some new core constructs such as parts, ports and connectors are introduced.

Part: The concept of parts makes possible the description of the internal structure of a class.

Port: The concept of a port makes it possible to describe connection points formally. These are addressable, which means that signals can be sent to them.

Connector: Connectors can be used to specify the communication links between two or more parts.

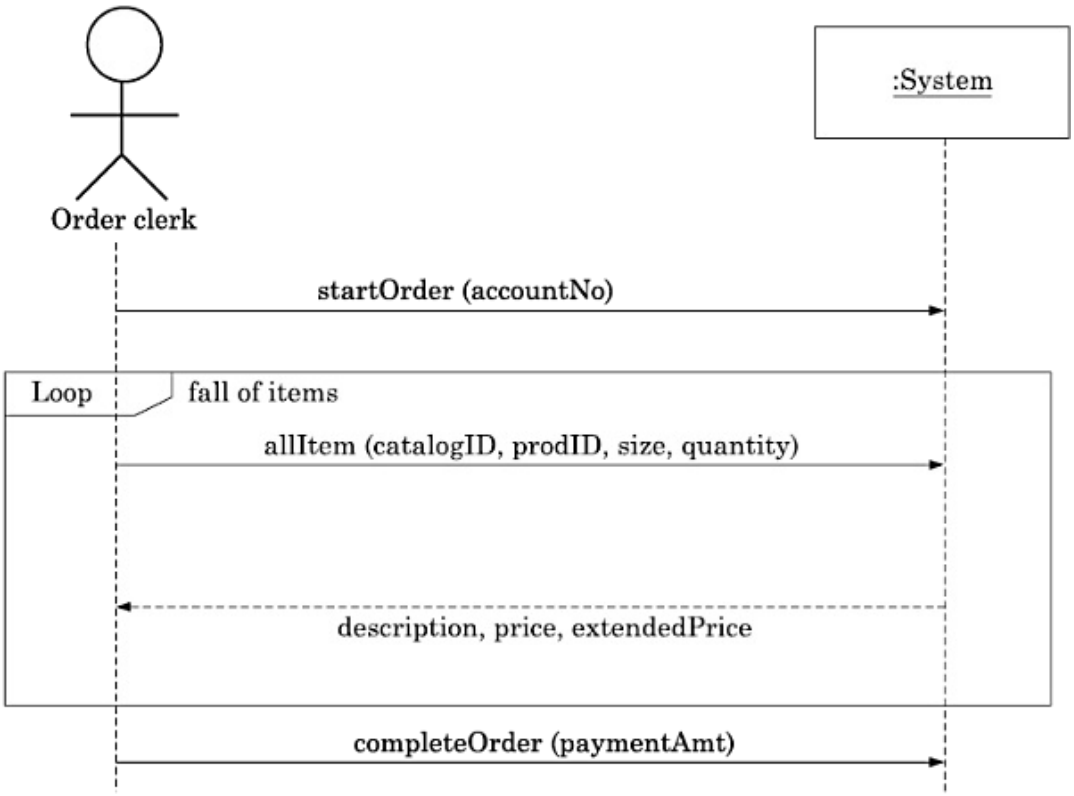

Figure 7.36: An example sequence diagram showing a combined fragment in UML 2.0.

SUMMARY

- In this chapter, we first reviewed some important concepts associated with object-orientation.
- One of the primary advantages of object-orientation is increased productivity of the software development team. The reason why object-oriented projects achieve dramatically higher levels of productivity can be attributed primarily due to reuse of predefined classes and partly reuse achieved due to inheritance, and the conceptual simplicity brought about by the object approach.
- Object modelling is very important in analysing, designing, and understanding systems. UML has rapidly gained popularity and is poised to become a standard in object modelling.
- UML can be used to construct five different views of a system using nine different kinds of diagrams. However, it is not mandatory to construct all views of a system using all types of diagrams in a modelling effort. The types of models to be constructed depends on the problem at hand.
- We discussed the syntax and semantics of some important types of diagrams which can be constructed using UML.

- In Chapter 8, we discuss an object-oriented system development process that uses UML as a model documentation tool.

EXERCISES

1. Choose the correct option:
   (a) The packing of data and functions into a single unit in a program is known as:
   (i) polymorphism
   (ii) abstraction
   (iii) encapsulation
   (iv) inheritance
   (b) Consider the statement—"An employee is either a worker or a manager." Assuming that Employee and Manager to be two classes, what can be said about the relationship between these two classes?
   (i) Association
   (ii) Generalisation-specialisation
   (iii) Containment
   (iv) Polymorphism
   (c) Which one of the following can be said about an abstract data type (ADT):
   (i) Same as an abstract class
   (ii) A data type that cannot be instantiated
   (iii) A data type that can only be used through the operations defined on it
   (iv) Same as a collection of data items
   (d) Which of the following can be said to represent the relationship between a class and its public parent class?
   (i) "...is a..."
   (ii) "...has a..."
   (iii) "...is implemented as a..."
   (iv) "...uses a..."
   (e) Which of the following sentences most closely describes "multiple inheritance"?
   (i) Where two classes inherit from each other
   (ii) When a base class has two or more derived classes
   (iii) When a child class has two or more parent classes
   (iv) When a child class has both an "is a" and a "has a" relationship with its parent

class

(f) Which one of the following best characterised inheritance?
(i) It is same as encapsulation.
(ii) Aggregation of information.
(iii) Generalisation and specialisation.
(iv) Polymorphism

(g) How is inheritance useful?
(i) It prevents inherited properties from being lost
(ii) It minimises the amount of code which has to be written
(iii) It creates elegant tree structures
(iv) It divides objects up into useful classes

(h) Consider the sentence: A book has one or more pages. Which of the following concepts characterise it best ?
(i) Inheritance
(ii) Specialisation
(iii) Association
(iv) Composition

(i) Which of the following indicated is a kind of relationship?
(i) Aggregation
(ii) Association
(iii) Dependency
(iv) Inheritance

(j) A sequence diagram is:
(i) A time-line illustrating a typical sequence of calls between object function members
(ii) A call tree illustrating all possible sequences of calls between class function
members
(iii) A time-line illustrating the changes in inheritance and instantiation relationships between classes and objects over time
(iv) A tree illustrating inheritance and relationships between classes
(v) A directed acyclic graph illustrating inheritance and instantiation relationships between classes and objects

(k) Which UML diagrams should you use when allocating use case behaviour to classes?
(i) sequence and communication diagrams
(ii) use case and activity diagrams
(iii) sequence and activity diagrams

(iv) class and composite structure diagrams

(l) Which of the following is a characteristic of a good object-oriented design:

(i) Deep class hierarchy

(ii) Large number of methods per class

(iii) Large number of message exchanges per use case

(iv) Moderate number of methods per class

(m) How do abstract and concrete classes differ from each other?

(i) Abstract classes represent intangible concepts in the application domain, whereas concrete classes represent physical things.

(ii) Abstract classes are superclasses, whereas concrete classes are subclasses. (iii) Abstract classes have no instances, whereas concrete classes have instances.

(iv) Abstract classes are a special type of concrete classes.

(n) Which one of the following false about encapsulation?

(i) Encapsulation helps in reuse since it is not necessary for other developers to know how a software component works internally.

(ii) Encapsulation means that software components can work more efficiently.

(iii) Encapsulation means that there is no need for software developers to document their work.

(iv) Encapsulation hinders reuse.

2. With the help of a suitable example explain how the inheritance feature of the object oriented paradigm helps in code reuse?
3. Can association relationship among classes be unary? If your answer is "yes", give an example of a unary association among classes. If your answer is "no", explain why such relationships cannot exist.
4. Is a class an abstract data type (ADT)? Justify your answer.
5. Give meaningful examples of each of the following types of relations among classes. Only class diagram and a line of explanation is required in each case.

(a) Single inheritance

(b) Multiple inheritance

(c) Association

(d) Aggregation

(e) Dependency

6. Consider the following sentences taken from various information descriptions to software development problems. From an analysis of the

sentences, identify the classes and relations among them that can be inferred from the sentences. Represent your answer using UML class diagrams:

(a) A square is a polygon
(b) Shyam is a student
(c) Every student has a name
(d) 100 paisa is one rupee
(e) Students live in hostels
(f) Every student is a member of the library
(g) A student can renew his borrowed books
(h) A college has many students
(i) A linked list consists of many nodes such that each node is a successor of some node and is the predecessor of some node.

7. With the help of a suitable example explain how polymorphism helps in developing easily maintainable and intuitively appealing code.
8. What do you understand by method overloading in the context of object-oriented programming? How is method overloading useful?
9. Can C++ and Java be considered as pure object-oriented programming languages? Justify your answer.
10. Explain the concept of dynamic binding as used in object-oriented languages using a simple illustrative example. How is dynamic binding useful program development?
11. What are the reasons behind the increased productivity that is noted when a development team adopts the object-oriented paradigm as against adopting a procedural paradigm?
12. Discuss the advantages and disadvantages of adopting an object-oriented style of software development.
13. In the context of object-orientation, distinguish between an operation and a method. Is it true that each operation must be implemented by a unique method?
14. What is the difference between method overloading and method overriding? Explain your answer by using a suitable example.
15. Inheritance and composition (object embedding) can be considered to be similar in the sense that both require a copy of the component(base) to be embedded(linked) in the compound(derived) object. Is it possible to use object embedding (i.e., composite objects) to realise the features of inheritance and vice versa ? Justify your answer by using suitable examples.

16. What do you understand by encapsulation and abstraction in the context of object- orientation? How is encapsulation any different compared to abstraction?
17. What is the difference between method overloading and method overriding? Explain the mechanisms of method overloading and method overriding using suitable examples.
18. What are the different system views that can be modelled using UML? What are the different UML diagrams which can be used to capture each of the views? Do you need to develop all the views of a system using all the modelling diagrams supported by UML? Justify your answer.
19. What is the difference between a use case and a scenario? Identify at least three scenarios of the withdraw cash use case of a bank ATM.
20. Why do you think UML requires several models from different perspectives to be constructed—would it not be a good idea to have just one model that captures all the required perspectives?
21. State **TRUE** o r **FALSE** of the following. Support your answer with proper reasoning:
    (a) An object oriented design cannot be implemented using a procedural programming language.
    (b) Any language directly supporting abstract data types (ADTs) can be called as an object-oriented language.
    (c) In contrast to an abstract class, a concrete class should define all its data and methods in the class definition itself without inheriting any of them.
    (d) An object-oriented language can be used to implement function-oriented designs.
    (e) The inheritance relationship describes the has a relationship among the classes.
    (f) Inheritance feature of the object oriented paradigm helps in code reuse.
    (g) Inheritance relationship between two classes can be considered as a generalisation- specialisation relationship.
    (h) Object embedding (i.e., composite objects) can be used to realise inheritance relationship.
    (i) Aggregation relationship between classes is antisymmetric.
    (j) The aggregation relationship can be recursively defined, i.e an object can contain instances of itself.
    (k) State chart diagrams in UML are normally used to model how some

behaviour of a system is realised through the co-operative actions of several objects.

(l) Multiple inheritance is the feature by which many subclasses can inherit features of one base class.

(m) Class diagrams developed using UML can serve as the functional specification of a system.

(n) An important advantage of polymorphism is facilitation of reuse.

(o) A static object-oriented model should capture attributes and methods of classes and in what order the different methods invoke each other.

(p) In a UML class diagram, the aggregation relationship defines an equivalence relationship an objects.

(q) Abstract classes and Interface classes (as used in UML, Java, etc.) are equivalent concepts.

(r) The aggregation relationship can be considered to be a special type of association relationship.

(s) The aggregation relationship can be reflexive.

(t) The aggregation relationship cannot be reflexive but is transitive.

(u) Normally, you use one interaction diagram per class to represent how the behaviour of an object of the class changes over its life time.

(v) It is possible that more than one methods in a class implement the same operation. (w) The chronological order of the messages in an interaction diagram cannot be determined from an inspection of the diagram.

(x) The interaction diagrams can be effectively used to describe how the behaviour of an object changes across execution of several use cases.

(y) From the UML sequence diagram for a use case, it would be possible to infer the various scenarios in the use case.

22. State **TRUE** o r **FALSE** of the following. Support your answer with proper reasoning:

(a) An object-oriented program that does not use the inheritance mechanism in the class definitions, cannot display dynamic binding.

(b) The inheritance mechanism can be thought of as providing feature abstraction.

(c) A state chart diagram is useful for describing behaviour that involves multiple objects cooperating with each other to achieve some behaviour.

(d) The implementation of a use case in terms of specific method calls is

depicted in a sequence diagram.

(e) The terms method and operation are equivalent concepts and can be used interchangeably.

(f) The effort to test and debug an object-oriented program can be reduced by reducing the number of message exchanges among objects.

23. What do you understand by the term encapsulation in the context of software design?
What are the advantages of encapsulation?

24. What do you understand by data abstraction? How does data abstraction help in reducing the coupling in a design solution?

25. What are the different types of relationships that might exist among the classes in an object-oriented design? Give examples of each.

26. What do you understand by association relation among classes. Give either real life or programming examples of unary, binary, and ternary association among classes.

27. What do you mean by an abstract class? Give an example of an abstract class. Abstract classes cannot have instances. What is then the use of defining abstract classes?

28. What are the different types of models of a problem that can be constructed using UML?
Why is it necessary to construct more than one type of model of a problem?

29. (a) Point out the main differences between an object-oriented language (e.g., C++) and a procedural language (e.g., C).
(b) Can an object-oriented design be implemented using a procedural language? Can a traditional function-oriented design be implemented using an object-oriented language? Write the reason behind your answer.

30. What basic features a programming language needs to support in order to be called as an object oriented language? How is an object oriented programming language different from a traditional procedural programming language such as C or PASCAL?

31. What is the difference between a use case and a scenario? Identify all scenarios of the withdraw cash use case of a standard bank ATM.

32. What is the difference between a sequence diagram and a collaboration diagram? In what context would you use each?

33. What is a stereotype in UML? Explain with some example situations

where these can be used?

34. How can you specify different constraints on the modelling elements in UML? For example, how can you specify that all books are kept alphabetically sorted in a library?
35. What is the difference between static and dynamic models in the context of object- oriented modelling of systems? Identify the UML diagrams which provide these two models respectively.
36. In modelling of systems using UML, how are the classification of users of a system into various types of actors and their representation in the use case diagram helpful in system development?
37. Draw a class diagram using the UML syntax to represent the fact that an orderRegister consists of many orders. Each order consists of up to ten order items. Each order item contains the name of the item, its quantity and the date by which it is required. Each order item is described by an item order specification object having details of an order item such as its unit price, name and address of the manufacturer, and the warranty period and terms of warranty.
38. Draw a class diagram using the UML syntax to represent the following aspects concerning a library. An issuable can either be a book or a CD. Books can be either reference books or text books. The details of various issuables are maintained in a register called the issuable register. The library has many members whose details are maintained in a member register. A member can issue upto 10 text books for a month. A member can also issue two CDs for a week.
39. Draw a class diagram using the UML syntax to represent the fact that the fleet of vehicles at a travel agency consists of vehicles of the types Tata Indica, Maruti van, and Mahindra Xylo. The regular customers of the travel agency can rent any vehicle they want. The details of the customers such as the name, address, and phone number are maintained by the agency.
40. Draw a class diagram using the UML syntax to represent the fact that the book register of a library contains details of the all the books in the library. The details for each book includes its title, author, ISBN number, price, date of procurement, price, and date of last loan, person to whom loaned. A book can either be a reference or issue type book. The reference books are to be referred inside the library and cannot be loaned out, whereas issue books can be taken on loan by a member. The member register contains the details of all members of the library.

The details that are maintained for a member include member name, address, telephone number, date of joining library and books outstanding. Each library member can take on loan at most five issue books.

41. Draw a class diagram using the UML syntax to represent the following. An engineering college offers B.Tech degrees in three branches—Electronics, Electrical, and Computer science and engineering. Each branch can admit 30 students each year. For a student to complete, B.Tech degree he/she has to clear all the 30 core courses and at least 10 of the elective courses.
42. Explain briefly how are the principles of decomposition and abstraction used in the object-oriented paradigm.
43. How is the activity diagram useful during system development? What are the important ways in which an activity diagram differs from a flow chart?
44. Mention the important shortcomings of UML 1.X. How has the UML 2.0 overcome these?
45. Develop the use case model for a word processor software such as MS-WORD.
46. Develop the use case model for a standard bank ATM.

1 A functional model captures the functions supported by the system.

2 An object model captures the objects in the system and their interrelations.

3 A more appropriate name for the stick person icon could have been 'role' rather than 'actor'. It appears that the apparent anomaly in the technical term used in referring to the stick person icon was caused by a wrong translation from the original Swedish document of Jacobson.

4 Many UML symbols are only suitable for drawing using a CASE tool and difficult to draw manually by hand. For example, italic names are very difficult to write by hand. When the UML diagram is to be drawn by hand, stereotypes such as <<abstract>> or constraints such as {abstract} can be used in place of difficult to draw symbols.

Chapter

8

OBJECT-ORIENTED SOFTWARE DEVELOPMENT

In this Chapter, we shall build upon the object-modelling concepts introduced in the last Chapter to discuss an object-oriented analysis and design (OOAD) methodology. We shall realise that object-oriented analysis and design (OOAD) techniques advocate a radically different approach compared to the traditional function-oriented design approaches. Recall our discussions in Chapters 5 and 6, where we had pointed out that the traditional function-oriented design approaches essentially suggest that while developing a system, all the functionalities that the system needs to support should be identified and implemented. In contrast, the OOAD paradigm suggests that the objects (i.e., entities) associated with a problem should be identified and implemented.

In the function-oriented design approach, a simple way to identify the functions performed by the system is to examine all the verbs occurring in the problem description—since verbs (such as create, edit, search, etc.) represent activities (or functions) performed by a system. Verb analysis is, in fact, an effective way to identify the functions of a system. On the other hand, the entities (or objects) occurring in a problem (such as book, member, register, etc. in a library automation software) can be identified by examining the nouns occurring in the problem description. Grady Booch summed up this fundamental difference between the function-oriented and object-oriented design approaches [1991] by saying:

... read the specification of the software you want to build. Underline the verbs if you are after procedural code, the nouns if you aim for an object oriented program.

Since the early nineties, a large number of object-oriented analysis and

design (OOAD) techniques have been proposed by researchers. Out of these, the most prominent ones possibly have been the Booch's approach by Grady Booch [1991], object modelling technique (OMT) by Rumbaugh and Blaha et al. [1991], Object-Oriented Analysis (OOA) by Coad and Yourdon[1991], and Objectory by Jacobson [1999]. All these OOAD techniques, in addition to identifying the different objects necessary to implement a system, also design the internal details of the objects. Further, the relationships existing among different objects are identified and represented in a design model, so that it becomes easy to code the design using a programming language. All OOAD methodologies essentially start by performing object-oriented analysis (OOA) to first develop an analysis model and then seamlessly refine this into a design model.

Object-oriented analysis (OOA) versus object-oriented design (OOD)

Before discussing the details of OOA and OOD, let us understand the primary differences between their intents and the performed activities.

The term object-oriented analysis (OOA) refers to developing an initial model of a software product from an analysis of its requirements specification.

Analysis involves constructing a model (called the analysis model) by analysing and elaborating the user requirements documented in the SRS document rather than determining how to define a solution that can be easily implemented. While developing the analysis model, implementation-specific decisions (such as in what sequence the classes would invoke each others' methods, specific hardware used, database used, etc.) are avoided. Therefore, the analysis model remains valid, even if the implementation aspects change later. However, it is very difficult to directly translate an analysis model into code. On the other hand, a design model can be easily translated into code. At present, many computer aided software engineering (CASE) tools support automatic generation of code templates from design models, thereby greatly reducing the programmer's work.

We now explain the difference between an analysis model and a design model through a simple example. Consider that one of the functionalities of a trade-house automation software is "display sales statistics," then the analysis model would involve representing various concepts identifiable in the description of this function such as the input data, the output data and the processing required. On the other hand, the design solution should address

how exactly (by which classes and method interactions) would the sales statistics be computed.

In the procedural approach, a marked difference is easily noticed between the analysis and design activities. The analysis activities concern developing the data flow model using DFD notation, whereas the design activities concern developing the design using the structure chart notation. Even the notations used to document the respective models[1] are different. In contrast, in the object-oriented approach, the transition from analysis to design activities is gradual and there is no clear demarcation between when analysis activities end and design activities start. Also, identical notations are used to document both analysis and design results.

An OOAD methodology

In this chapter, we shall discuss a generic methodology for developing object-oriented designs starting from initial problem descriptions. This methodology consists of first constructing a use case model from the initial problem analysis. Subsequently, the domain model is constructed. These two analysis models are iteratively refined into a design model. The design model can straight away be implemented using a programming language. It must, however, be kept in mind that though the design methodology that we shall discuss is easy to master, it is useful only to solve simple problems. However, once we are able to understand this simple design method, approaches for solving more complex problems can be comprehended with only incremental effort.

This chapter is organised as follows. We first discuss design patterns. We subsequently introduce an object-oriented analysis and design methodology which is to a large extent based on the unified process [Jacobson 1999] and the work of Rosenberg [2000]. Subsequently, we illustrate the working of the discussed methodology through a few examples.

8.1 PATTERNS

The concept of a pattern was first originated in the field of Architecture, where large buildings are designed in specific pattern solutions. This concept has now been absorbed in the area of object-oriented design to provide a very powerful mechanism for design reuse.

Design patterns and their role in OOAD

While working out the design solution to a problem, experienced

designers consciously or unconsciously reuse solutions that they might have worked out in the past. Such reuse of the design solutions is systematised by the concept of patterns. In fact, patterns allow commonly accepted solutions to be reused by everybody familiar with the patterns. We shall soon see that while working out the design solution to a problem, a knowledge of some important design patterns can go a long way to improve the design quality and at the same time reduce the total effort.

But, what are design patterns?

> Design patterns are commonly accepted solutions to some problems that recur during designing different applications.

Design patterns are nowadays being used extensively and have been found to make the design process efficient. Use of patterns reduces the number of design iterations, and at the same time improves the quality of the final design solution. This makes a thorough study of the patterns worthwhile. Once we become familiar with a few important patterns, we can spot them in the designs we try to work out and shall be able to reuse the pattern solutions.

8.1.1 Basic Pattern Concepts

Every non-trivial problem consists of a large number of subproblems. Solving a problem therefore involves solving all its subproblems. In fact, while solving any two problems, several subproblems that are common between the two can be identified. Further, a few subproblems repeat across a large number of problems. This opens up the possibility arriving at good solutions with reduced efforts by mastering the commonly accepted solutions to a few important subproblems that repeat across different problems. This in fact, captures the central idea behind patterns.

In the context of software design, patterns document solutions to certain problems that are reusable during the designs of different applications. While working out the solution to a design problem, once we identify a pattern, we can straightaway use the documented pattern solution, if we have understood it well. We can now state the basic idea behind patterns as follows:

> The basic idea behind patterns is that if you can master a few important patterns, you can easily spot them in application development problems and effortlessly use

the pattern solutions.

As we shall see, pattern solutions do not really espouse any revolutionary design ideas, but are created based on sound common sense and application of the fundamental design principles. You might then wonder "If patterns are really based on commonsense, then what is the point learning patterns? Wouldn't one mechanically arrive at similar solutions any way?" In reality, it is observed that while grappling with the nitty gritty of complex design problems, designers often forget commonsense—and tend to make their design solutions unnecessarily complex, inflexible, and inefficient. On the other hand, if you are familiar with a few important patterns, you can easily spot them and mechanically use the pattern solutions in your designs. Thus, a pattern serves as a guide for creating a "good" designs. Also, use of patterns significantly increases the productivity of the designers, by reducing design iterations and at the same time improving the quality of the final design. In view of the inherent advantages, experienced designers usually make themselves familiar with hundreds of patterns.

Patterns can be viewed as helping designers to make certain important design decisions. At a basic level, patterns can also be viewed as well-documented and well thought-out building blocks for software design.

But, who creates the patterns? Patterns are created and documented by people who spot repeating themes across designs. Once patterns are created and documented, they can be used by different designers. Therefore, we can consider patterns as a highly effective means to capture and transfer design knowledge across different design problems. Patterns also have come to form a standard vocabulary that is used for communicating design ideas in a professional environment.

In addition to providing model solutions, patterns document clear specifications of the problems, and also explain the circumstances in which a solution would work and would not work. A pattern documentation usually consists of four important parts:

- The problem.
- The context in which the problem occurs.
- The solution.
- The context within which the solution would work and would not work.

In the following subsections, we discuss a few important concepts about patterns. In the subsequent section, we provide an overview of a few important patterns.

8.1.2 Types of Patterns

Different types of patterns have been identified for use in different stages of design. Starting from use in very high-level designs termed as architectural designs, pattern solutions have been defined for use in concrete designs and even in code.We discuss some basic concepts about these three types of patterns as follows.

Architectural patterns: The architectural patterns identify and provide solutions to problems that are identifiable while carrying out architectural (or very high-level) designs. Architectural designs concern the overall structure of software systems. Architectural designs cannot directly be translated to code, but form the basis for more detailed design. Each architectural pattern suggests a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organising the relationships among them. Architectural designs are usually constructed for very large problems. Naturally therefore, architectural patterns are relevant while working out the high-level solutions to very large problems.

Design patterns: A design pattern usually suggests a scheme for structuring the classes in a design solution and defines the required interactions among those classes. In other words, a design pattern describes some commonly recurring structure of communicating classes that can be used to solve some general design problems. Design pattern solutions are typically described in terms of classes, their instances, their roles and collaborations. In this text, we restrict ourselves to design patterns only and discuss a few important design patterns in Section 8.2

Idioms: Idioms are a set of low-level patterns that are programming language-specific.

> An idiom describes how to implement a solution to a particular problem using the features of a given programming language.

In any natural language such as English, an idiom means a group of words that together have a meaning that is different from the one obtained by a simple juxtaposition of the dictionary definitions of the individual words. Examples of English idioms are "raining cats and dogs," "at a stone's throw," etc. Idioms help to write good pieces of prose with substantially reduced efforts. While composing a prose, the linguists effortlessly incorporate idioms into their writing to improve the quality of the prose and at the same time achieve reduction in the time and effort taken to compose the prose. When a

linguist wants to say that it is raining very heavily, he almost mechanically writes "It is raining cats and dogs." without really having to think of the specific word combinations to use: such as whether to say "It is raining mouses and elephants", etc. Similarly for accessing the different elements of an array of size 100, a C programmer would write the code segment: `for(i=0;i<100;i++){...}` without having to think and decide whether to use a while loop, whether to start the loop with the variable i initialised to 1, etc. Good programmers know several idioms. As soon as they see a requirement while working out a programming solution, the required idiom occurs to them instantly and relieves them from having to struggle to select of the exact constructs to use in the program and at the same time improves the quality of the program and reduces the bug detection and correction iterations.

Comparison of different types of patterns

The main differences among the three kinds of patterns discussed above lie in the levels of abstraction and details they deal with. Architectural patterns are high-level strategies that concern the overall solutions to large-scale problems. Design patterns are solutions for specific parts of medium-scale problems and recommend certain structures and behaviour of the participating entities. Idioms are paradigm-specific and language-specific programming solutions that recommend using appropriate code segments for solving low-level programming problems. In this text, we restrict ourselves to design patterns only.

8.1.3 More Pattern Concepts

We now discuss a few other important pattern concepts in the following subsections.

Patterns versus algorithms

Beginners often confuse between patterns and algorithms and ask questions such as—"Are patterns and algorithms identical concepts? After all, both target to provide reusable solutions to problems!" In fact, patterns and algorithms are in some respects similar since both attempt to provide reusable solutions. However, algorithms primarily focus on solving problems with reduced space and/or time requirements, whereas patterns focus on understandability and maintainability of design and easier development.

In contrast to algorithms, patterns are more concerned with aspects such as

maintainability and ease of development rather than space and time efficiency.

Pros and cons of design patterns

Before using design patterns during OOAD, it is desirable to be familiar with the pros advantages and cons disadvantages of design patterns.

The following are the main pros strengths of design patterns:

- Design patterns provide a common vocabulary that helps to improve communication among the developers.
- Design patterns help to capture and disseminate expert knowledge.
- Use of design patterns help designers to produce designs that are flexible, efficient, and easily maintainable.
- Design patterns guide developers to arrive at correct design decisions and help them to improve the quality of their designs.
- Design patterns reduce the number of design iterations, and help improve the designer productivity.

Important cons shortcomings of design patterns are the following:

- Design patterns do not directly lead to code reuse. Since a design pattern is tailored for a specific circumstance of reuse, and therefore it is difficult to associate a fixed code segment with a pattern.
- At present no methodology is available that can be used to select the right design pattern at the right point during a design exercise.

Antipattern

If a pattern represents a best practice, then an antipattern represents lessons learned from a bad design. The following are two types of antipatterns that are popular:

- Those that describe bad solutions to problems, thereby leading to bad situations.
- Those that describe how to avoid bad solutions to problems.

Antipatterns are valuable because they help us to recognise why a particular design alternative might seem at first like an attractive solution, but later on lead to complicacies and finally turn out to be a poor solution. After we become familiar with the important antipatterns, we can consciously try to avoid them while solving a problem.

We mention here only a few interesting antipatterns without discussing them in detail, since we feel that they are out of the scope of this text.

Input kludge: This concerns failing to specify and implement a mechanism for handling invalid inputs.

Magic pushbutton: This concerns coding implementation logic directly within the code of the user interface, rather than performing them in separate classes.

Race hazard: This concerns failing to see the consequences of all the different ordering of events that might take place in practice.

8.2 SOME COMMON DESIGN PATTERNS

It is important to become familiar with at least the important patterns, since after you understand these patterns well, you should be able to spot these patterns while solving problems and then you would be able to effortlessly use the commonly accepted solutions captured by the patterns. Also, a familiarity with the pattern terminologies would help you to master the vocabulary needed to discuss alternate pattern solutions with your colleagues while working out design solutions. For example, your colleague might while reviewing your design suggest a way to improve your design by saying "Why not use MVC pattern in the part of your design concerned with handling user inputs?"

It should be remembered that patterns offer generic solutions that usually need to be fine tuned in the context of specific problems. We, however, take the liberty of discussing design patterns in the context of solutions to specific design problems for ease of understanding. Though a large number of design patterns have been proposed so far, the ones by Larman and gang of four or GoF (proposed by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides) are extremely popular. We now describe a few important design patterns from these authors.

EXPERT

Problem: When a certain activity needs to be performed, which class should be made responsible for doing it?

Solution: Assign responsibility to the information expert—the class that has all (or most of) the information necessary to fulfill the required responsibility. The expert pattern expresses the common intuition that objects should do things related to the information they store. The class and collaboration

diagrams for the solution to a problem as to which class should compute the total cost of a sale transaction is shown in Figure 8.1.

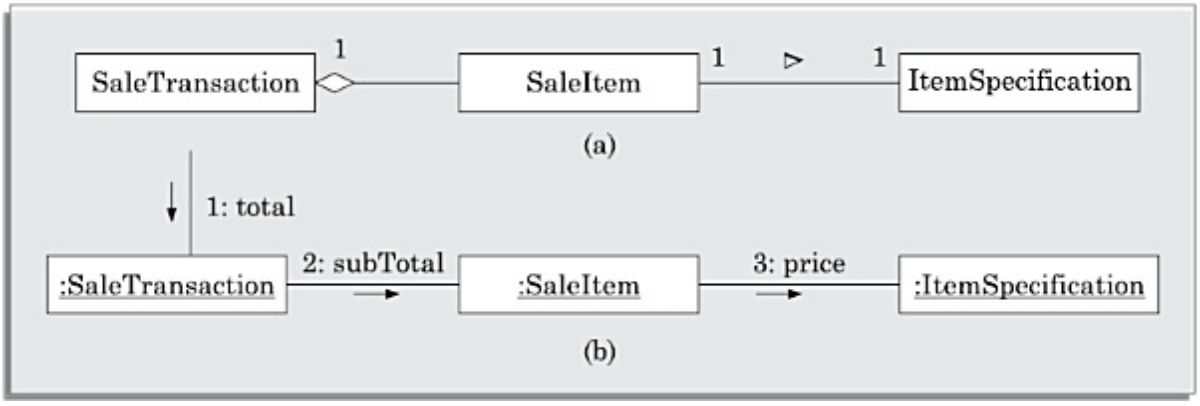

Figure 8.1: Expert pattern: (a) Class diagram (b) Collaboration diagram

Explanation: In the example given in Figure 8.1, a sale transaction consists of many sale items. Each saleItem object is associated with an itemSpecification object. The itemSpecification object among other things indicates the unit price for the item. In this situation, which object should compute the total price for a sale transaction? Let us consider the different options that are available. Should the saleTransaction object, the saleItem object, or the itemSpecification object be given the responsibility to compute the price of a sale transaction? If we assign the responsibility to the saleItem object to compute the transaction price, then it would need several information from the saleTransaction object such as the number of items sold for various items and could also need the prices for the other items from the corresponding itemSpecification objects for this computation. This would require a large number of data exchanges among the objects to occur, and lead to a poor quality solution. Similarly, itemSpecification would be a poor choice for computing the total cost payable for a sale transaction, since the object lacks most of the information required to compute the total cost of the transaction. It can easily be seen that the saleTransaction object has most of the information required for the computation, and should be assigned the responsibility to compute the total price. It is the information expert, as far as the exact items and quantities sold are concerned.

CREATOR

Problem: Which class should be responsible for creating a new instance of a certain class?

Solution: Assign a class C1 the responsibility to create an instance of class C2, if one or more of the following are true:

- C1 is an aggregator of objects of type C2.
- C1 contains objects of type C2.
- C1 closely uses objects of type C2.
- C1 has the data that would be required to initialise the objects of type C2, when they are created.

Explanation: An aggregator object needs to create all its component objects, since the aggregator needs to maintain the addresses of its component objects. Any other object needing some information from a component would have to request the aggregator to retrieve the information. Occasionally, in an object-oriented programming problem, an object may need to be created which is not a part of any aggregate object. In this case, the object should be created by the one that has the required initialization parameters, or the one that would work most closely with the object.

Facade pattern[2]

Problem: How should the services be requested from a service package by client classes (i.e., classes outside the package)?

Context in which the problem occurs: A package, as already discussed in Chapter 7, is a cohesive set of classes. That is, the classes in a package have strongly related responsibilities. For example, an RDBMS interface package would contain classes to provide services that an application programmer can invoke to perform various operations on the RDBMS.

Solution: A separate class (such as DBfacade) should be created to provide a common interface to the services provided by the classes in the package.

Explanation: Let us understand the necessity and importance of a facade class using a simple analogy. Consider a lecture hall complex. Suppose all students are required to enter the lecture hall complex through the main entrance only. In this case, when some general information such as cancellation of some lecture on a certain date is to be brought to the notice of all the students, displaying one notice at the entrance should be enough. But, consider the situation where the students enter the lecture hall complex through multiple entrances, and are even allowed to jump into the class

rooms through the different windows of various class rooms. In this case, notices have to be displayed at all possible points of entry of the students. Similarly, in the context of a service package, when (the type or number of) the parameters of some method of a class in the package changes, all classes that invoke the method have to be changed. However, when outside objects (clients) invoke services through a facade class, the change can be absorbed in the facade class and changes to the client classes can be avoided. Also, use of a facade class reduces the complexity of service invocation, since client classes need not be aware of the specific classes that implement the services. The discussed advantages of a facade class can be easily observed from the schematic representations shown in Figure 8.2.

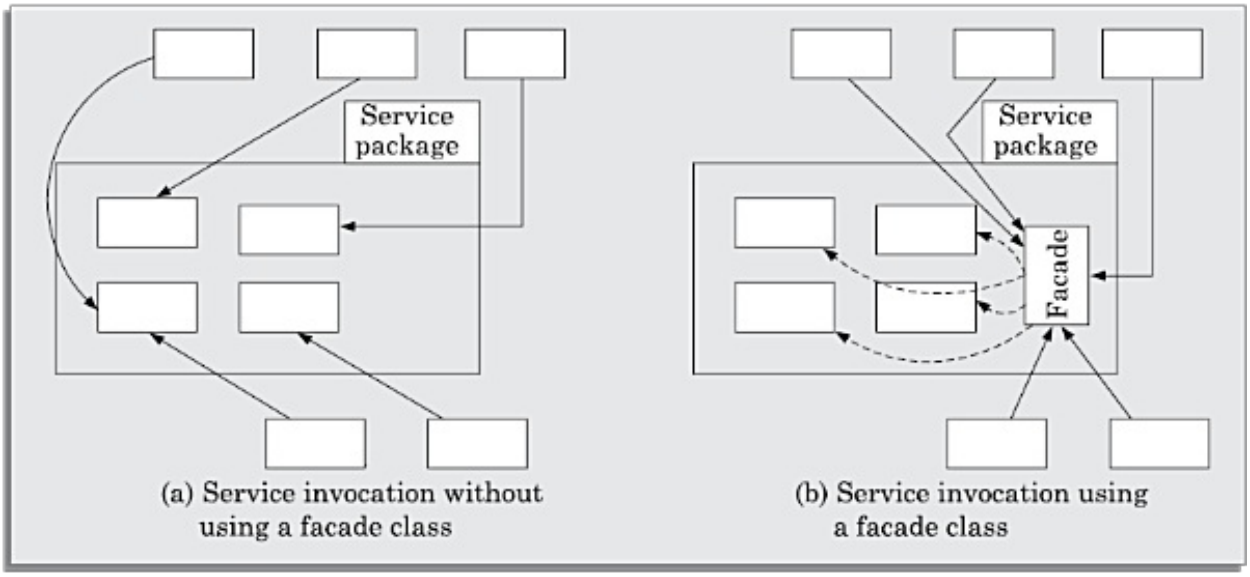

Figure 8.2: Service invocation with and without using a facade class.

MODEL VIEW SEPARATION PATTERNS

Problem: How should the non-GUI classes communicate with the GUI classes and vice versa?

Context in which the problem occurs: This is a very commonly occurring pattern which can be identified in almost every design problem. Here, view is a synonym for the presentation layer objects (or GUI objects). and model is a synonym for the domain layer (non-GUI) objects, The domain layer objects are responsible for actually providing the required service, whereas the presentation layer objects are responsible for handling only the interactions (input/output) with the user.

Solution: Depending on the context, the solution provided by either of observer pattern, model-view-controller(MVS)pattern, or the publish-

subscribe pattern (discussed subse- quently) can be used.

Explanation: Whenever any interaction between a model object and a view object is required, the model object should be loosely coupled to the view objects. This would help to make reuse of the model objects easy. Also, the view objects normally undergo changes much more frequently as compared to the model objects. Each change to a view object should not require changes to be made to the model objects.

When information is required to be displayed synchronously, a pull from above solution is satisfactory. However, this solution does not work satisfactorily for asynchronous information display. When information is produced asynchronously by a model object, the view object would not know when exactly would new information be available so that request for it can be made. Also, polling (periodically requesting information to check if new information has become available) is not a satisfactory solution on efficiency considerations.

There are many situations in which information is required to be displayed asynchronously. Consider a software that monitors the stock market quotations continuously and alerts the user when the required stock quotations reach some prespecified threshold value. This software would pop-up an alert message as soon as any of the stock quotation reaches its threshold value. In this case, the alert message pops up asynchronously. The GUI would not be able to pull this information since it won't know the precise time when a threshold would be reached. There are numerous other examples where such asynchronous display is required, and include a network intrusion monitor, a computer fault monitor, etc.

The main idea behind the model-view separation pattern is to achieve loose coupling between the model and view objects, so that the complexity of design is reduced and reuse of the model objects is enhanced. There are several variations of model-view separation pattern that are useful in different situations of interaction between the model and the view objects. In the following subsection, we discuss the following three important model-view separation patterns.

OBSERVER PATTERN

Problem: When a model object is accessed by several view objects, how should the interactions between the model and the view objects be structured?

Solution: Observers should register themselves with the model object. The

model object would maintain a list of the registered observers. When a change occurs to the model object, it would notify all the registered observers. Each observer can then query the model object to get any specific information about the changes that it might require. This pattern therefore uses both the push and the pull modes. The interaction diagram for this pattern is shown in Figure 8.3.

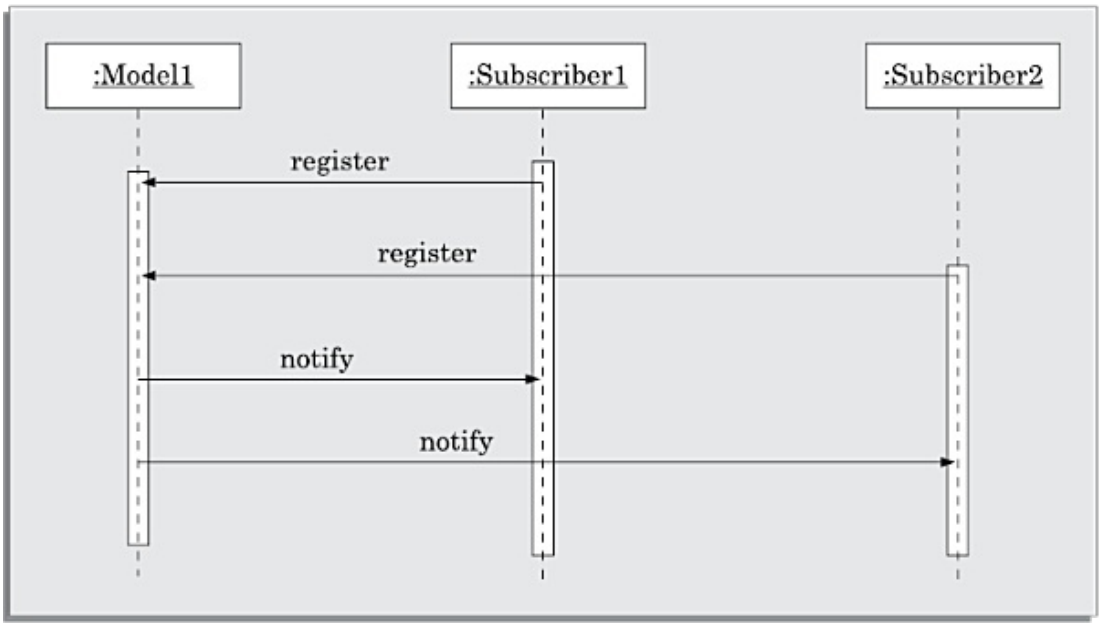

Figure 8.3: Interaction diagram for the observer pattern.

Explanation: The observer pattern solution achieves loose coupling between the model and observer objects. Also, it achieves decoupling among the observers themselves, they need not be aware of each other. Once an observer has been informed of a change, it may query the model for some specific piece of information.

The observer pattern has the following limitations:

- The model objects incur a substantial overhead to support registration of the observers, and also to respond to the queries from the observers. This is detrimental to performance of the model objects, especially when the number of observers is large.
- The two-stage process (notification and query) reduces coupling between the model and observer objects. However, selective notification to only the interested observers would make this solution very inefficient, since unnecessary message exchanges may occur when different observers are interested in different types of events.

MODEL-VIEW-CONTROLLER (MVC) PATTERN

Problem: How should the GUI objects interact with the model objects?

Solution: The GUI objects need to be separated into view and controller types. The controller objects are responsible for collecting the data input by a user. The controller would pass the collected information onto the model object. The model object would notify the view and controller objects regarding the change in model state. The class and collaboration diagrams for the MVC pattern have been shown in Figures 8.4 and 8.5 respectively.

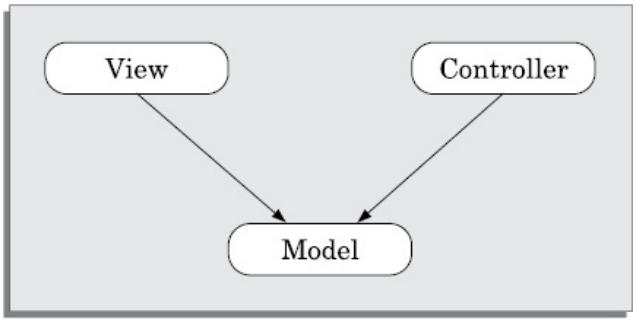

Figure 8.4: Class structure for the MVC pattern.

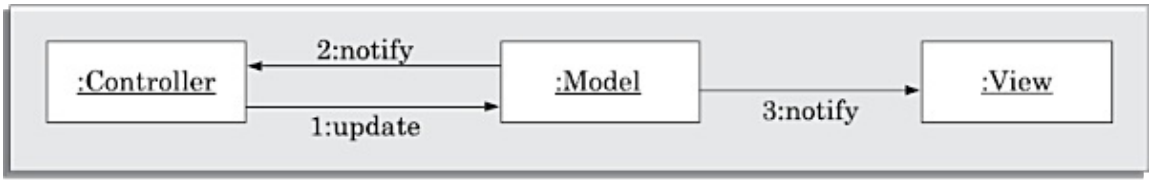

Figure 8.5: Interaction model for the MVC pattern.

Explanation: When a controller object collects input data and passes those onto a model object, the state of the model object may change. When the state of the model changes, it can send an update message to all dependent view and controller objects. The purpose of the model object notifying the view object is obvious (that is, to refresh the display with the updated information). The controller object also needs to be notified, since the new model object state may need dimming certain menu options, or even closing certain control objects. Benefits of this pattern include loose coupling between the view and model objects and a clear separation between the code that handles inputs, that displays data, and that processes the application's data.

This is useful in situations where multiple views of the same data needs to be presented. In particular, this pattern is useful in applications in which the model object can change its state asynchronously and multiple consistent views of the model object need to be displayed effectively. As a simple example, the MVC pattern can be used when for the same input data several

different representations such as line plot, bar chart, and pie chart representations need to be shown. In this case, whenever the model changes its data or properties, all dependent views are automatically updated.

PUBLISH-SUBSCRIBE PATTERN

The publish-subscribe pattern is a more general form of the observer pattern and overcomes many of the shortcomings of the observer pattern.

Problem: When a given model object is accessed by a large number of view objects, and the model state changes asynchronously, how should the interaction be structured?

Solution: This pattern suggests that an event notification system should be implemented through which the publisher (model objects) can indirectly notify the subscribers as soon as the necessary information becomes available. An event manager class can be defined which keeps track of the subscribers and the types of events they are interested in. An event is published by the publisher by sending a message to the event manager object. The event manager notifies all registered subscribers usually via a parameterised message (called a callback).

The publish-subscribe pattern has been schematically shown in Figure 8.6. Observe that the GUI objects have subscribed to the event `e1` with the corresponding event manager. As soon as a model object publishes the event `e1`, the corresponding event manager would notify (callback) the GUI objects that have subscribed to the event.

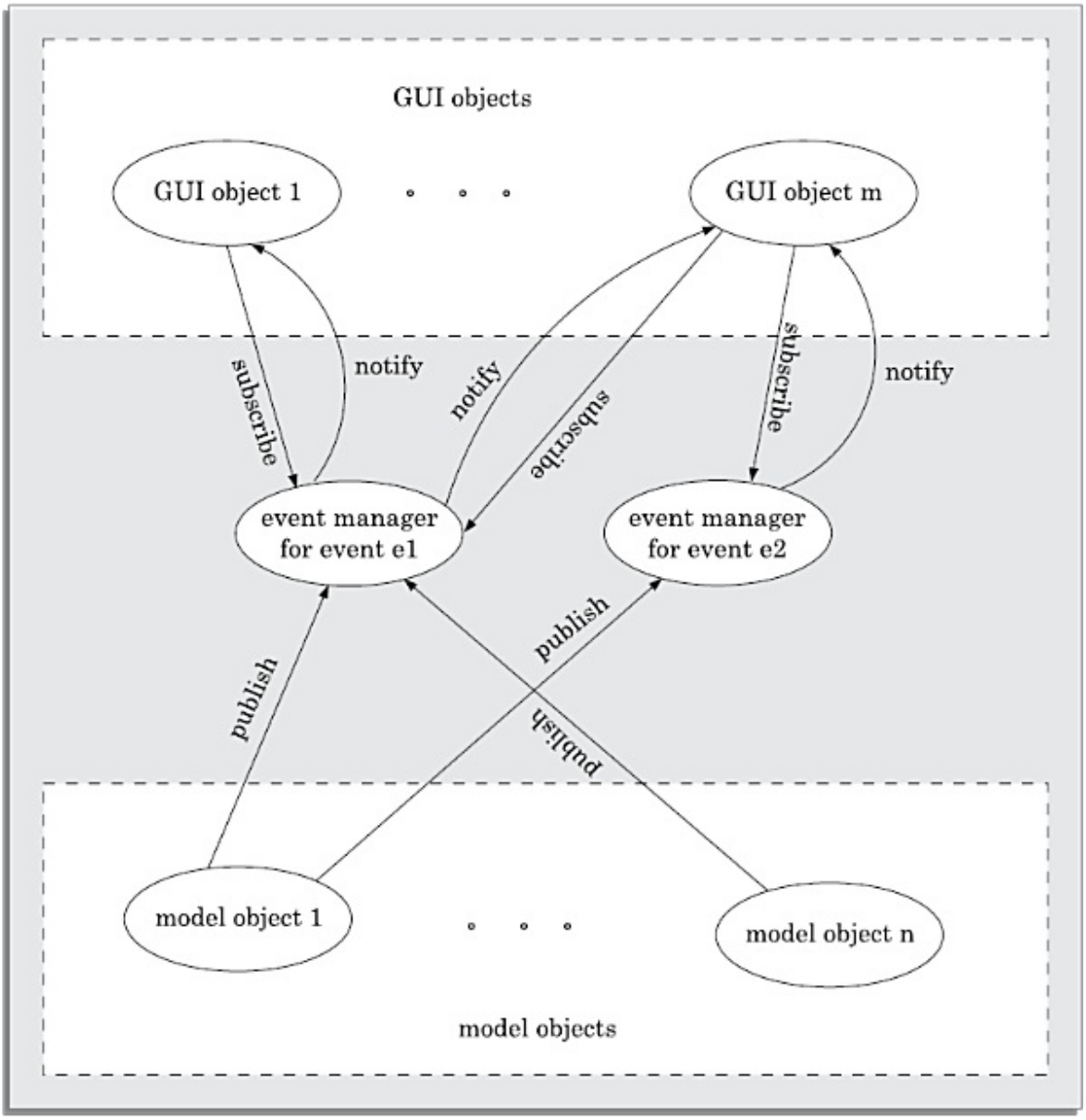

Figure 8.6: Interaction model of the publish-subscribe pattern.

Explanation: The the publish-subscribe pattern requires creating an event manager class for each type of event that the model objects may create. A subscriber may register its interest for a class of events by subscribing to the corresponding event manager classes. Events generated by publishers are selectively notified to subscribers. The selection is achieved by defining event manager classes. As soon as an event occurs, the event manager passes the information to all those who have subscribed to the event. For example, when a publisher generates an event `e1`, the corresponding event manager is notified of the same. All subscribers that previously subscribed to the event `e1` would receive an asynchronous notification of the occurrence of `e1`. The subscriber may respond to the event by associating a specific handler with

each class of events. The event manager is responsible for registering subscriptions, receiving events from models, filtering events, and passing them to the interested subscribers (see Figure 8.7). Depending on the specific system, the event manager may be implemented in various ways, e.g., by a centralised server, by distributed co-operating servers, or possibly collectively by the publishers. The interaction diagram is shown in Figure 8.7.

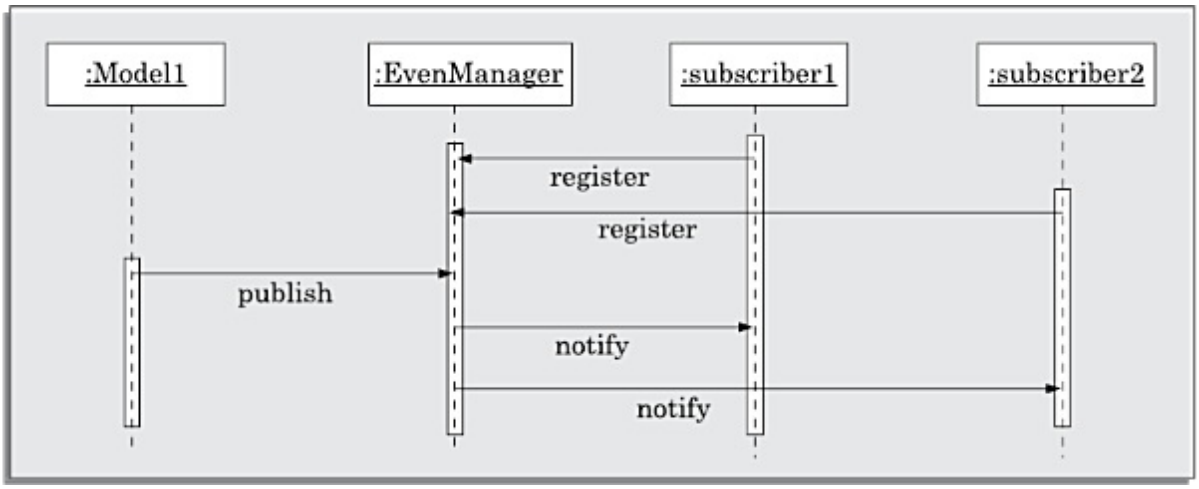

Figure 8.7: A schematic representation of the publish-subscribe pattern.

Compared to the observer pattern, this pattern frees the model object from handling the registration of observer objects and notification objects. Thus, this pattern has obvious advantage over the observer pattern, especially when the number of observers is large. Modern object-oriented languages support several event manager classes. For example, Java provides the EventListener interface for such purposes.

INTERMEDIARY (OR PROXY) PATTERN

Problem: How should a client object invoke the services of a server object?

Context in which the problem occurs: The terms client and server refer here to the objects existing across a network. The clients are consumers of services and the servers are the providers of services.

Solution: A proxy object should be created at the client side. The proxy object would act as local sit-in for the remote server object. The client should make all its service requests to the proxy, the proxy would in turn transmit the service request to the appropriate server, obtain the response, and deliver it to the client object.

Explanation: The proxy hides the details of the network transmission. The proxy is responsible for determining the server address, communicating the

client request to the server, obtaining the server response and seamlessly passing that to the client. The proxy can also augment (or filter) information that is exchanged between the client and the server. For example, a proxy may compress or encrypt a message while sending it to the server and uncompress or decrypt a message on receipt. The proxy should have the same interface as the remote server object, so that a client can feel that it is interacting directly with the remote server object. Thus in effect, the proxy object hides (abstracts out) the complexities of network transmissions.

8.3 AN OBJECT-ORIENTED ANALYSIS AND DESIGN (OOAD) METHODOLOGY

The design process that we are going to discuss here starts with the analysis activities. The results of the analysis activities are be refined into a design model through several iterations. Considering that the unified process is very popular for object-oriented software development, we first set the context regarding the phase of the unified process in which the discussed design methodology is applicable. An overview of the analysis and design methodology is presented in Section 8.3.2. We subsequently discuss the analysis and design process in more detail and finally work out solutions to a few example problems to illustrate its use.

8.3.1 Unified Process

Unified process is incremental in iterative process model for object-oriented software development that has gained acceptance among the practitioners and academicians. The first book to describe the unified process was titled "The Unified Software Development Process" and was published in 1999 by Ivar Jacobson, Grady Booch and James Rumbaugh. Since then, various authors unaffiliated with the erstwhile Rational Software Corporation have published books and articles using the name unified process, whereas authors affiliated with Rational Software Corporation have favoured the name rational unified process (RUP).

The unified process is an extensible framework which needs to be customised for specific types of projects.

The two main characteristics of the unified process are: use case-driven and iterative.

The term use case-driven implies that use cases (customer's view) of the

system is considered to be the central and most important view. The use case view should be the first one to be constructed and should be refined iteratively into an implementation.

The use case model is the central model. All models that are constructed in the subsequent design activities must conform to the use case model.

As shown in Figure 8.8, the unified process involves iterating over the following four distinct phases as follows:

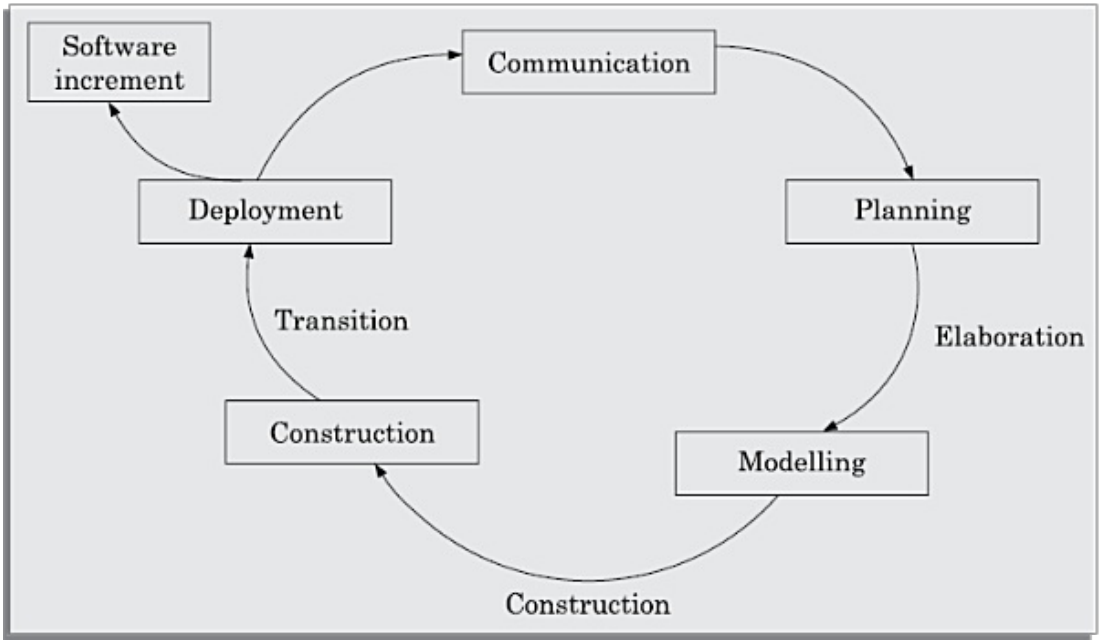

Figure 8.8: Unified process model.

Inception: During this phase, the scope of the project is defined and prototypes may be developed to form a clear idea about the project.

Elaboration: In this phase, the functional and the non-functional requirements are captured. The preliminary use case and the domain model are developed during this phase.

Construction: During this phase, the design and implementation activities are carried out. Full text descriptions of use cases are written and each use case is taken up for the start of a new iteration. System features are implemented in a series of short iterations and are tested. Each iteration results in an executable release of the software.

Transition: During this phase, the product is installed in the user's environment and maintained.

The design process that we have discussed in Section 3.2 is undertaken during the elaboration and construction phases of the unified process.

8.3.2 Overview of the OOAD Methodology

The object-oriented analysis and design (OOAD) methodology that we are going to discuss has schematically been shown in Figure 8.9. As shown in Figure 8.9, the use case model is the first model to be developed. As pointed out in Section 8.3.1, in any user-centric development process such as the unified process, all developed models must conform to the use case model. Therefore, the use case model plays a crucial role in the design process, and needs to be developed first. As shown in Figure 8.9, the domain model is constructed next through an analysis of the use case model and the SRS document. The domain model is refined into a class diagram through a number of iterations involving the interaction diagrams. Once the class diagram has been constructed, it can easily be translated to code. Many CASE tools support generation of code skeleton from the class diagram.

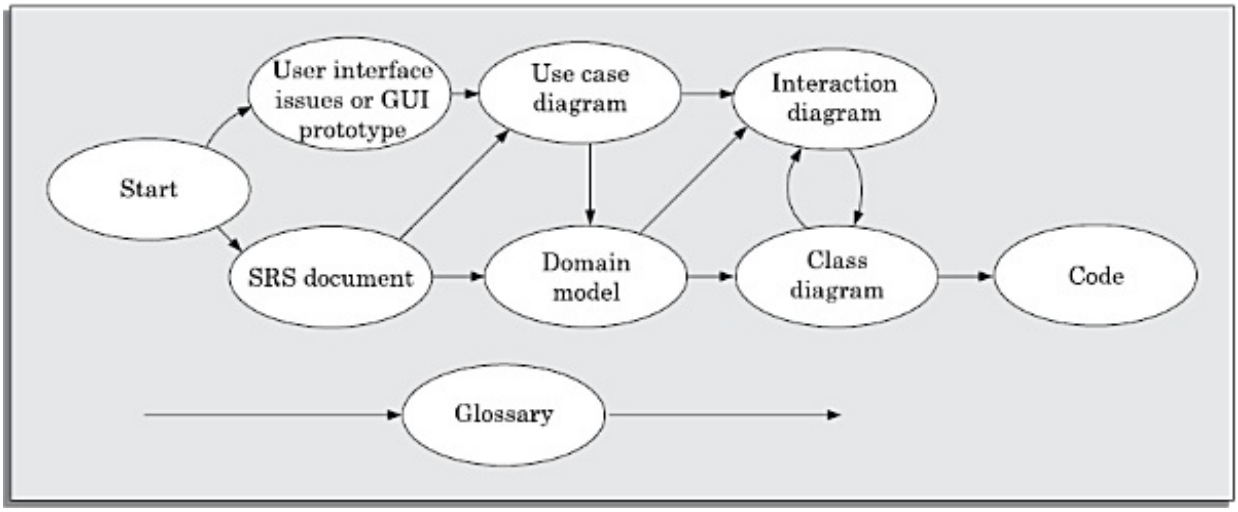

Figure 8.9: An object-oriented analysis and design process.

Throughout the analysis and design process, a glossary is continually and consciously created and maintained. A glossary is a dictionary of terms which can help in understanding the various terms (or concepts) used in the constructed model. The terms listed in the glossary are essentially concept names. The glossary (or model dictionary) lists and defines all the terms that require explanation in order to improve communication and to reduce the risk of misunderstanding. Maintaining the glossary is an ongoing activity through out the project as shown in Figure 8.9.

8.3.3 Use Case Model Development

We had pointed out in Chapter 7 that a use case model, in essence, captures the high-level user requirements of the system to be developed. For every

use case, the details of all the scenarios of user interactions with the system are captured in an accompanying text description. In this section we address how should one go about to develop the use case model for a given problem? The use cases can easily be identified from the SRS document. In fact, the high- level functional requirements normally correspond to the use cases. But, it often turns out that most practical problems have too many use cases. Therefore, these use cases have to be appropriately packaged. However, how can one determine the package structure of use cases? It is worth noting that there is a close correspondence with the structure of the GUI, the organisation of the users' manual into sections, and the packaging of the use cases.

An overriding principle while identifying and packaging use cases is that there should be a strong correlation between the GUI prototype, the contents of the users' manual and the use case model of the system.

We explain what this correlation usually means through an example. Consider a text editor software (such as the openOffice, MS-Word, or Word-perfect, etc.) that can be used to create various kinds of text documents. The top-level menu options for such a word processing software would be File operations, Edit, View, Insert, Tools, etc. Each of these high-level menu options can be considered to be a container for a set of functions and usually represented using a package in the use case diagram. For example, the File menu may contain functions such as—load file, store file, save, save as, print, etc. The Edit menu may contain the functionalities for cut, paste, select, group, etc.

Each of the menu options in the top-level menu of the GUI would usually correspond to a package in the use case diagram.

A few of the functions under a menu option can themselves contain functions in the sense that clicking them would open up a submenu. Such submenu would correspond to a package in the first-level packages in the use case diagram.

The way use cases are grouped into packages also has a strong correlation with the organisation of the users' manual into chapters and sections. The users manual would can exactly the same chapters as the first-level packages of the use case diagram (or the top-level menu organisation in the GUI). The sections of a chapter would correspond to the packages in the corresponding first-level package, etc. As an example, for the case of the word processing software, the different chapters of the users' manual would be arranged into

File operations, Edit, View, and Tool operations, etc., corresponding to the GUI menu structure and also the use case packages.

Considering the close correspondence of the users' manual and the GUI prototype with the use case organisation, it can be advantageous to develop the users' manual first and then develop the use case model based on this. This provides a convenient summary of what use case-driven development means.

Normally, carrying out the GUI prototyping in parallel with the development of the SRS document (as shown in Figure 8.9) is considered an excellent approach. This involves iterating the presentation aspects of the system with the user. After achieving closure on different screens, the corresponding use case model can be developed. Even though a close correspondence should exist between the GUI prototype and the use case model, the use cases should not be too tightly tied to the GUI. For example, the use cases should not make any reference to the type of the GUI element appearing on the screen, e.g., radioButton, pushButton, etc. This is necessary because, the type of the user interface components used may change frequently during software development and more so during later maintenance. However, the functionalities provided by the system do not change so often.

A use case is most effectively named from the perspective of the user. It should be named using present tense verb phrases, and should be in active voice, e.g., `admit patient`, `issue book`, `generate report`, etc., rather than `patient admission`, `book issual`, `report generation`, etc.

Common mistakes committed in use case model development

The following are some common mistakes that beginners commit during use case model development. We are listing these mistakes with the hope that you will consciously try not to commit these and the related types of mistakes while developing use case models:

Clutter: Too many use cases at the top-level use case diagram make it very difficult to understand the model. When large number of use cases are present in the top-level of the use case diagram, they should be organised into packages. Packages are an effective way to manage complexities. Each use case package should correspond to one chapter or section of the users' manual or a top-level menu choice in the GUI.

Too detailed: Often beginners confuse substeps of use cases with separate use cases. For example, One should may inappropriately define a use case print receipt, whereas it should only be a sub-step of the withdraw cash use case. A use case model in which the subfunctions of a high-level function are represented as full use cases would make the later analysis and design tasks difficult.

Omitting text description: Omitting text description of use cases makes it very difficult for any one to gain full understanding of the use case behaviour and also makes it very difficult to design the system.

Overlooking some alternate scenarios: It is necessary to capture all alternate scenarios of each use case. Overlooked scenarios can later show up as missing functionalities or bugs.

8.3.4 Domain Modelling

Domain modelling is also known as conceptual modelling. A conceptual model depicts the concepts[3] (or objects) that are easily identifiable from the problem description. A domain model usually contains three types of objects—objects that correspond to physical entities in the problem description, objects that would handle the user interface (also called boundary objects), and objects that are entirely conceptual (also called controller objects).

The objects identified during domain analysis can be classified into three types:
- Boundary objects
- Controller objects
- Entity objects

Examples of physical (entity) objects in a library automation software are book, book register, member register, library member, etc. Examples of a conceptual object and a boundary object can be issue book controller, and issue book user interface, respectively. A domain model should also capture the relationships that may easily be identified among these objects.

A domain model can be considered to be the first-cut class diagram and is obtained from an analysis of the problem description. In a domain model, no methods or attributes are associated with classes and only the names of the classes are represented. The methods (responsibilities) and data (attributes) are usually not represented. The methods and attributes are identified and represented later in the design process.

In the following subsections, we will discuss the important characteristics of these three different types of objects that are identified during construction of

a domain model.

> While boundary and controller objects can be mechanically identified from an inspection of the use case diagram, identification of entity objects requires practice. So, we can say that the crux of the domain modelling activity is to identify the entity objects judiciously.

We first discuss how the boundary and conceptual object are identified from an inspection of the use case diagram. Subsequently, we discuss a methodology to identify the entity objects.

Boundary ob jects

The boundary objects are those with which the actors interact. The boundary objects include screens, menus, forms, dialogs, etc. The boundary objects are mainly responsible for evaluating user interactions through suitable graphical user interfaces (GUIs). These objects normally do not include any processing logic. However, they may be responsible for validating inputs, formatting outputs, etc. We can say in other words that the main responsibilities of a boundary object can be to read inputs from the user, validate the inputs, format the outputs, and display the results.

The boundary objects were earlier being called as the interface objects. However, the term interface class is now being used with a different meaning (as pointed out in Chapter 7) for UML and also for Java and COM/DCOM.

> The initial identification of the boundary classes can be made by defining one boundary class per actor/use case pair.

You might wonder that when two or more different actors are associated with a use case (for example, the `register customer` use case in Figure 8.16) why should two different boundary classes be used? The reason behind this is that different categories of users have different privileges and different levels of familiarity with the software package. For example, the clerk would everyday use the interface and would need an efficient interface rather than a very user-friendly but slow interface.

Later on during the design process, a boundary class may be split into two or more classes if it is found to be performing a large and complex set of tasks. Two boundary classes may be combined into a single one, if they have similar responsibilities.

Entity ob jects

The entity objects normally hold information such as data tables and files (e.g., Book, `BookRegister`, `LibraryMember`, `etc.)`. Among all the three types of classes we discussed, the entity classes are the only classes that can store data permanently or semi-permanently. The entity objects usually store data across use case executions and therefore need to outlive use case executions. Usually the entity objects act like "dumb servers". This means that these objects carry out only simple processing on their stored data such as update, store, search, retrieve, etc. Such primitive operations on data usually do not change often with time. Therefore, entity objects undergo much less changes during the operation phase of the software compared to the other types of objects and are said to be more stable than the other types of objects.

A popular methodology to identify the entity classes is discussed in Section 8.3.5.

Controller ob jects

Typically, every use case execution involves several interactions among a group of objects. Each object involved in the execution of a use case plays its part (performs certain actions) to help complete the execution of the use case. In this context, usually a controller object co-ordinates the activities of a set of collaborating objects to deliver the results corresponding to an actor request.

For every use case, a separate controller object should be created and given the responsibility to handle actor requests.

For every invocation of a use case by the same (or different) actor, a separate controller object should be instantiated from the corresponding controller class. When an actor invokes a use case, the same controller object should handle all the interactions with that actor This way, it becomes possible to easily maintain the necessary information about the state of execution of the use case by an actor. The state information maintained by a controller can be used to identify the out-of-sequence actor requests (e.g., whether the print voucher request is received before arrange payment request), and it can then take appropriate actions. The name "controller" is appropriate since this object co-ordinates (or controls) the activities of several objects for serving a user request.

A controller object typically orchestrates the activities of a set of entity

classes to implement the behaviour of a use case and also manages the user interactions through a set of boundary classes (see Figure 8.10). From Figure 8.10 it can be seen that the controller accepts inputs and displays outputs to the users through two boundary classes (Boundary 1 and Boundary 2), it also co-ordinates the activities of the entity classes (Entity 1, Entity 2, and Entity 3) to realise the behaviour associated with the use case.

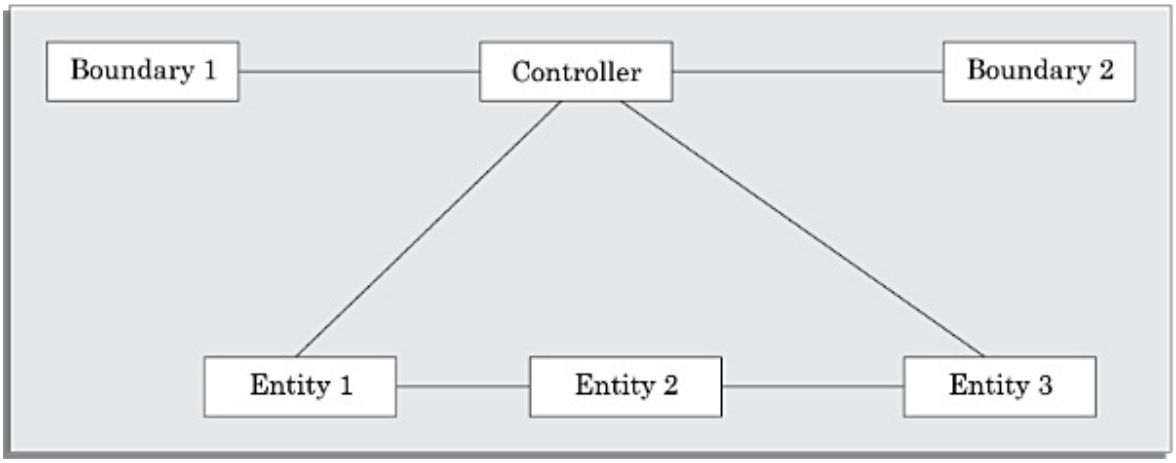

Figure 8.10: A typical realisation of a use case through the collaboration of boundary, controller, and entity objects.

A few questions that may arise in our minds concerning the controller classes can be as follows:

What is the advantage of using a controller class?

A controller object effectively decouples the boundary and entity objects from one another, reducing design complexity and making the system tolerant to changes of the user interface and processing logic. The controller objects embody most of the business processing logic required for the realisation of the use case (the business logic may change from time to time). For this reason, we can like a controller object with a manager who commands and oversees the activities of a set of workers. When the business logic changes, only the controller and boundary objects may change, entity objects do not.

What is the role of a controller ob ject?

For every use case, a controller object is given the charge of realising the behaviour associated with the use case by making use of the services of the required entity and boundary classes. For example, the controller of the `renewBook` use case in a library automation system may first request to retrieve and supply the exact books that have been borrowed by the user, it

would then ask the user to make a selection of the books that need to be renewed through the boundary class. It would then request the `BookRegister` to check whether any one has reserved the requested books, in such a case it would inform the user that it would not be possible to renew the book. Otherwise, it would request the `BookRegister` to renew the book and display the information to the user. Thus, the controller is the intelligent class that knows the exact subtasks to be completed to deliver the required results when a client invokes a use case. The controller orders different objects to complete the different subtasks. In the OOAD terminology, a controller class is said to "realise the corresponding use case."

> The controller class can be considered as the intelligent class and the other (entity and boundary) classes as the dumb classes that are driven by the controller class.

Is exactly one controller classes required per use case?

As we have already mentioned, each use case is normally realised using one controller object. However, very simple use cases can be realised without using any controller object, i.e., through boundary and entity objects only. This is often true for trivial use cases that perform only some simple operations that can be done either by the boundary class itself or with the help of a single entity class. More complex use cases that need to implement significant business logic may require more than one controller object to realise the use case. A complex use case can have several controller objects with responsibilities such as transaction manager, resource co-ordinator, and error handler. For every invocation of a use case, a separate instance of the corresponding controller class needs to be created. For example, the use cases require the controller object to transit through a number of states. Therefore, the exact state of controller for every instance of execution of a use case needs to be tracked. In such cases, one controller object might have to be created for each execution of the use case.

In our methodology, in the initial solution, one controller class is created for every use case. Later, if it is determined that a controller class has very superficial or trivial responsibilities, it can be omitted from the design. If it is found to be too complex, it might be split into a number of controller classes.

What if some classes are missed in the domain model?

For creating a domain model, the recommended strategy is to quickly create a rough conceptual model where the emphasis is on finding the

obvious concepts (objects) expressed in the requirements while deferring a detailed investigation. Of course, for complex problems, it is possible that some classes can be missed by the designer. Later during the design process, the conceptual model is incrementally refined and extended. In this process, some classes that were missed in the class model would be identified and incorporated, and some of the classes that have too many responsibilities may be split into multiple classes, and even some classes not considered necessary may be deleted. The methods and the attributes of the classes as well as the class relationships would be established and added later on.

8.3.5 Identification of Entity Ob jects

In any object-oriented design methodology, identification of entity objects is a crucial step. In fact, the quality of the final design depends to a great extent on the appropriateness and the completeness of the identified entity objects. However, to date, no systematic step-by-step methodology exists for identification of entity objects, though the boundary and controller classes are easily identified and can be determined almost mechanically from an analysis of the use case diagram. Several semi-formal and informal approaches have been proposed for identification of the entity objects. All these techniques require application of common sense and several subjective judgments need to be made based on past experience while applying the object identification techniques. Various object identification techniques can be classified into the following broad classes:

- Grammatical analysis of the problem description.
- Derivation from data flow.
- Derivation from the entity relationship (E – R) diagram.

A widely accepted object identification approach is the grammatical analysis approach that was proposed by Grady Booch[1991]. In Booch's grammatical analysis approach, the nouns occurring in the extended problem description (processing narrative) are mapped to objects and the verbs are mapped to methods. The extended problem description is the function (black box) description of the input data, output data, and the processing that needs to be done on the input to get the output. The approaches for identification of entity classes based on derivation from the data flow diagram and entity-relationship model can be used to refine the results obtained using the Booch's object identification methodology.

8.3.6 Booch's Ob ject Identification Method

Booch's object identification approach requires a processing narrative of the given problem to be first developed. The processing narrative is an overall description of the problem and also a discussion on how the given problem can be solved. Out of the objects identified from the processing narrative, if an object is concerned with implementing a solution, then it is said to be a part of the solution space. Otherwise, if an object is necessary only to describe the problem, then it is said to be a part of the problem space. The objects are identified through lexical analysis of the processing narrative by noting down the nouns (name words) in the processing narrative. Of course, many of the nouns listed from the processing objective may not be objects. From this list of nouns, synonym of a noun should of course be straight away eliminated.

It should be noted that several of the nouns may not correspond to objects. The following are some criteria that can be used to eliminate some of the nouns that have been identified through a grammatical analysis of the processing narrative and converge on the actual set of objects:

Users: There can be various categories of users (actors) of a software and these users normally appear as name words in the problem description. The actors themselves and the interactions among the actors should be excluded from the entity identification exercise. However, sometimes there may be a need to maintain information about an actor within the system. Only in such cases, classes corresponding to the names of the actors should be considered. Such classes sometimes are called surrogates. For example, in the library information system (LIS) we would need to store information about each library member such as his name, address, identification number, phone number, etc. This is independent of the fact that the library member also plays the role of an actor of the system.

True name word: An imperative procedure name, i.e., noun form of a verb actually represents an action and should not be considered as an object. For example, cash withdrawal in the processing description of an ATM refers to withdrawing cash and is an action and not a noun, and therefore can be deleted from the list of nouns.

Retained information: Every entity object must have some attributes (stored data) associated with it and the methods of the entity object should operate on this data. It would be very unusual if we have an object that has

only a set of methods and no retained information. These retained information is the private data of the object that we discussed earlier. If an object does not contain any private data, it cannot normally be expected to be an entity object. Thus, if we cannot associate any meaningful attribute with an object, we should eliminate it from the list of nouns. For example, consider the following statement in a processing narrative—The results are displayed on computer screen. Here, computer screen cannot be an object since we cannot associate any information to be stored on a computer screen for later processing.

Multiple attributes: Usually objects have multiple attributes and support multiple methods. It is very rare to find useful objects which store only a single data element or support only a single method, because an object having only a single data element or method is usually implemented as the part of another object. Thus, if we cannot associate multiple attributes and methods with a noun, we should conveniently exclude it from the list of nouns. Consider the statement: "Each student has a phone number." Here, a phone number should not be made an entity object since we cannot associate any information other than the phone number with the phone object.

Common operations: Once we determine a set of operations that can be identified for one potential object, these operations should apply to all occurrences of the same object. Only then should we define the noun as a class. We have already seen that an attribute or operation defined for a class must apply to each instance of the class. If some of the attributes or operations apply only to some specific instances of the class, then we need to have one or more subclasses for these special objects. Consider the statement—"Resident students need to be assigned a hostel room, whereas for non-resident students the local contact phone number and the local address needs to be stored." Here, we need two classes: resident student and non-resident student that need to be derived from a student class, since these two types need to have different attributes and operations, though some operations and attributes would be similar.

Although the grammatical approach is simple and intuitively appealing, yet through a naive use of this approach, it is very difficult to achieve high quality results if the approach is used naively. In particular, it is very difficult to come up with useful abstractions simply by doing grammatical analysis of the problem description. Useful abstractions usually result from clever factoring of the problem description into independent and intuitively correct elements. However, the grammatical approach can serve as a rough guide that one

needs to use with some thought rather than by letter.

Other helpful guidelines

The following are a few more helpful guidelines to identify entity objects from the processing narrative:

Aggregate objects: Entity classes normally occur as aggregate objects. This is because the data to be stored is typically distributed among several objects. For example, student objects may be aggregated into a student register, book objects may be aggregated into a book register, etc.

Correspond to data stores in DFD: Often, the entity objects roughly correspond to the data stores in a well-designed DFD. Thus, the DFD model, if available, can help identify the entity objects.

Registers in physical world: The entity objects usually correspond to the registers that needed to be maintained in the manual working of the system, For example, in a library, normally book registers and member registers are maintained. These can be considered as corresponding to aggregate entity objects.

Example 8.1 Let us identify the entity objects of the following **Tic-tac-toe software:**

Tic-tac-toe is a computer game in which a human player and the computer make alternate moves on a 3 × 3 square. A move consists of marking a previously unmarked square. A player who first places three consecutive marks along a straight line (i.e., along a row, column, or diagonal) on the square wins. As soon as either the human player or the computer wins, a message congratulating the winner should be displayed. If neither player manages to get three consecutive marks along a straight line, and all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game.

If we perform a grammatical analysis of this problem statement, we will come up with the nouns that have been italicised. However, on closer examination we can eliminate the synonyms from the identified nouns. The list of nouns after eliminating the synonyms are the following—Tic-tac-toe, computer game, human player, move, square, mark, straight line, board, row, column, and diagonal.

From this list of possible objects, we can eliminate nouns like human player as it is the actor. Also, we can eliminate the nouns square, game, computer, Tic-tac-toe, straight line, row, column, and diagonal, as we cannot associate

any data and methods with them. We can also eliminate the noun move from the list of potential objects since it is an imperative verb and actually represents an action. Thus, we should be left with only one meaningful object —board.

Once we become a little experienced in object identification after solving a few problems, it would not normally necessary to really identify all nouns in the problem description by underlining them or actually listing them down, and systematically eliminating the non-objects to arrive at the final set of objects. We can with some experience identify the set of objects by carefully reading the problem description and analysing it mentally.

8.3.7 Interaction Modelling

Recall from our discussions in Chapter 7 that the behaviour associated with a use case is realised through the interaction of several objects. These interactions are co-ordinated by the controller object. The interactions occurring among a group of objects to realise a use case is captured through an interaction diagram. The primary goals of interaction modelling are the following:

- To allocate the responsibility of a use case realisation among the boundary, entity, and controller objects. The responsibilities for each class is reflected as an operation (method) to be supported by that class.
- To show for each use case the detailed interaction that occur over time among the associated objects to complete the execution of the use case.

We had already seen in Chapter 7 that interaction modelling is captured through UML sequence and collaboration diagrams. However, collaboration diagrams can be obtained mechanically from sequence diagrams. Therefore, it is usually sufficient to develop the sequence diagrams alone. You normally need to do one sequence diagram for each use case. That is, for each use case (consisting possibly of many scenarios) only a separate sequence diagram should be designed to capture the interactions occurring among the associated objects. Later in the design process, an interaction diagram can be split into two or more separate diagrams, if the interaction diagram gets too complicated or cannot be accommodated legibly on a standard size paper. In such cases, the use case would have to be split into simpler use cases as

discussed in Section 7.4.4, and the interaction diagrams done for the simplest use cases.

8.3.8 Class-Responsibility-Collaborator (CRC) Cards

Class-Responsibility-Collaborator (CRC) approach was pioneered by Ward Cunningham and Kent Beck at the research laboratory of Tektronix at Portland,Oregon,USA. The interaction diagram of a simple use case usually involves capturing the collaboration among a few objects. For such simple use cases, the interaction diagrams can easily be drawn from an inspection of the use case description. However, designing interaction diagrams for more complex use cases may involve collaboration of many objects and the interactions among these objects can be difficult to comprehend for an individual. Developing the interaction diagram for such use cases may require participation of a team of developers through the use of CRC cards.

> In the CRC card approach, a number of team members participate to assign responsibilities to the classes involved in a use case realisation.

CRC cards are essentially index cards that are used to assign responsibilities to classes. One CRC card is prepared for each class represented in the domain model. On each of these cards, the responsibility of each class is written in brief as and when they are identified. The objects with which this object needs to collaborate for fulfilling its responsibility are also written.

CRC cards are usually developed in small group sessions where people role play being various classes. Each person holds the CRC card of the classes he is playing the role of. The cards are deliberately made small (4 inch ×6 inch) so that each class can have only limited number of responsibilities. A responsibility is the high level description of the part that a class needs to play in the realisation of a use case A responsibility would normally be a single method, but may also be implemented by several methods. The different use cases are taken up one by one. Starting from the initiation of the use case by the actor, the interactions with the user are analysed and the specific tasks that need to be performed by the system are determined. The team members role-playing for the classes determine if their class should take up the responsibility by considering the pros and cons. An example CRC card for the `BookRegister` class of the Library Automation System is shown in Figure 8.11.

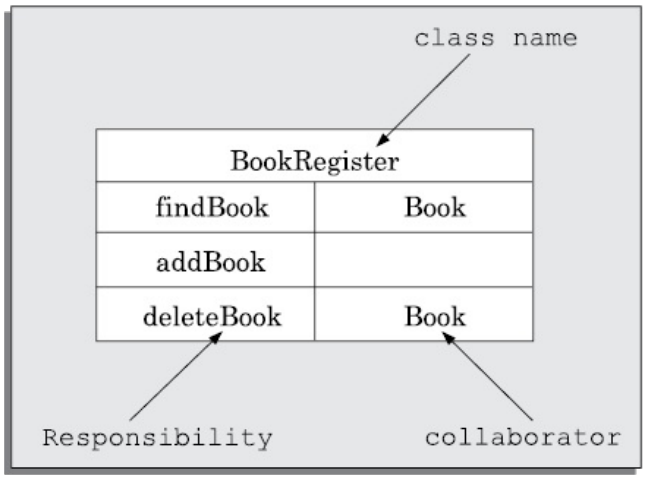

Figure 8.11: CRC card for the BookRegister class.

Once you have assigned the responsibility to classes using CRC cards, you can develop the interaction diagrams by flipping through the CRC cards.

8 . 4 APPLICATIONS OF THE ANALYSIS AND DESIGN PROCESS

We now demonstrate how the analysis and design process can be used by applying them on two example problems.

Example 8.2 Consider the Tic-tac-toe computer game outlined in Example 6.2 of Chapter 6. A step-by-step workout of this example is as follows:

- The use case model is shown in Figure 8.12.

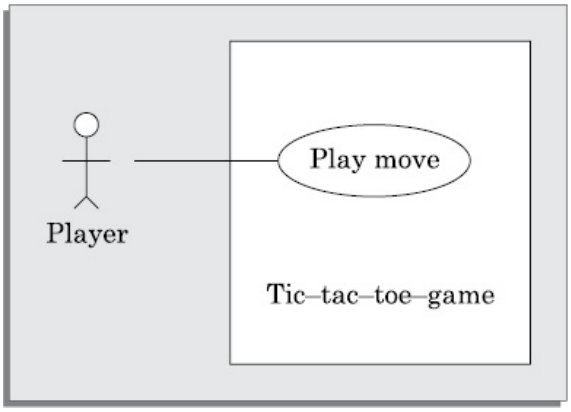

Figure 8.12: Use case model for Example 8.2.

- The initial domain model is shown in Figure 8.13(a).
- The domain model after adding the boundary and control classes is shown in Figure 8.13(b).

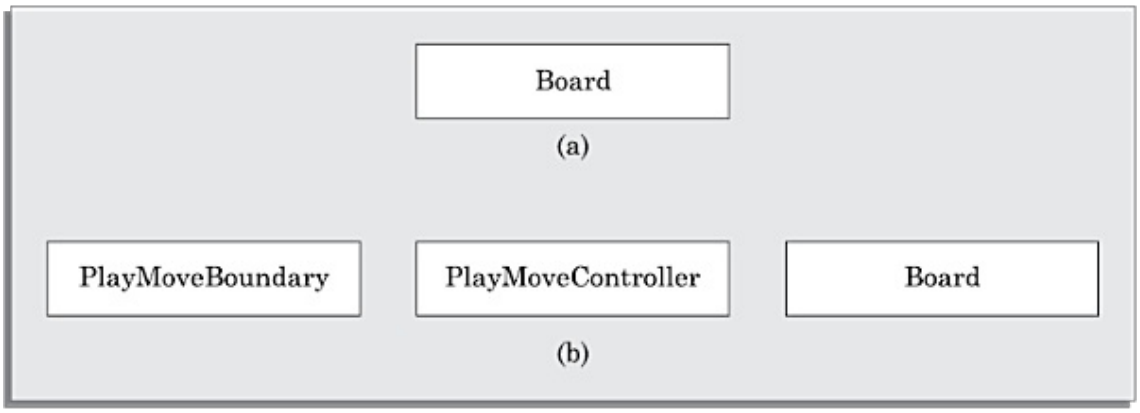

Figure 8.13: (a) Initial domain model (b) Refined domain model for Example 8.2.

- Interaction diagram for the play move use case is shown in Figure 8.14. Note that the expert pattern is used while determining which class should be responsible for (i.e., contain) the `checkWinner` method, the controller or the boundary class? The Board class has all the necessary information, based on which the winner can be determined. Therefore the board class has been made to support the `checkWinner` method.

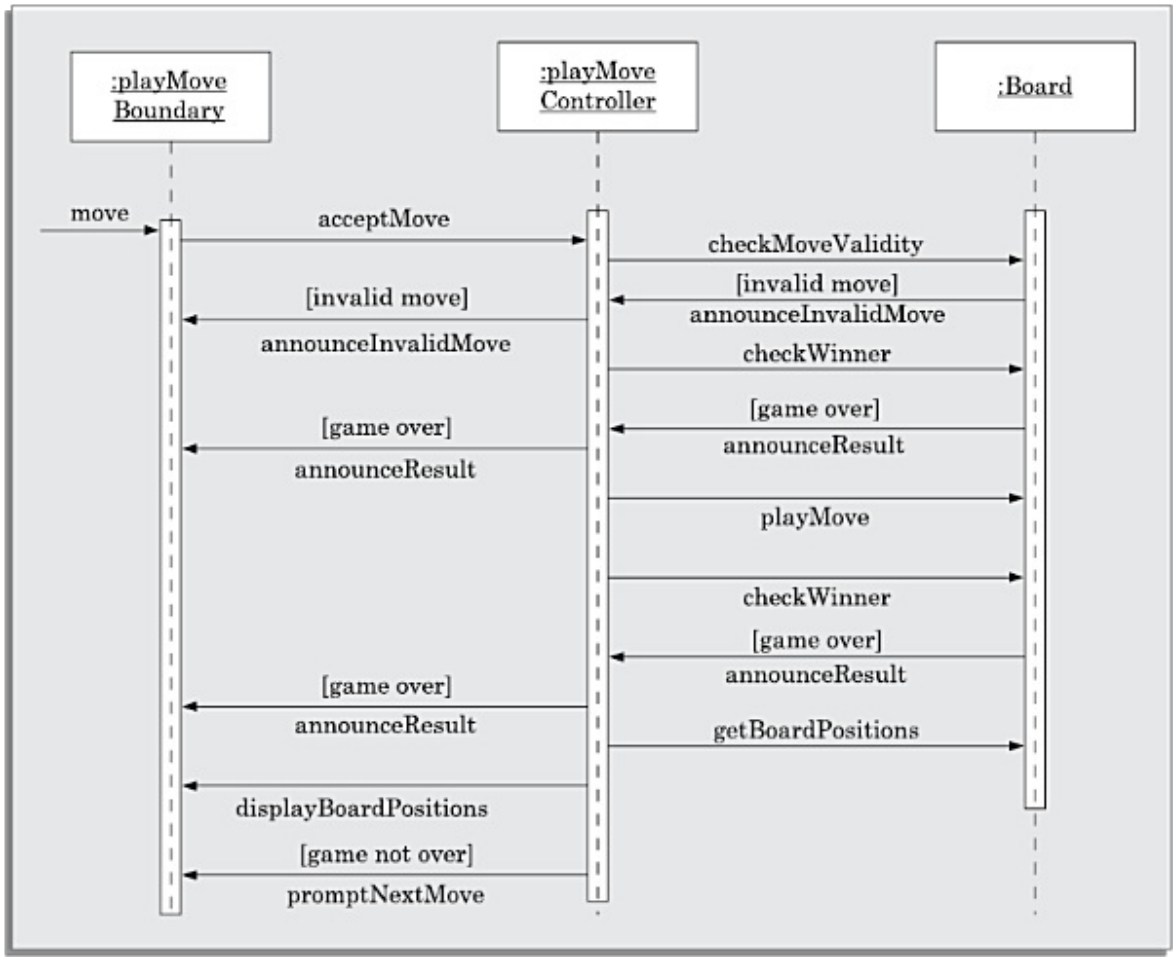

Figure 8.14: Sequence diagram for the play move use case of Example 8.2.

- Class diagram is shown in Figure 8.15. The messages of the interaction diagram of the play move use case has been populated as the methods of the corresponding classes.

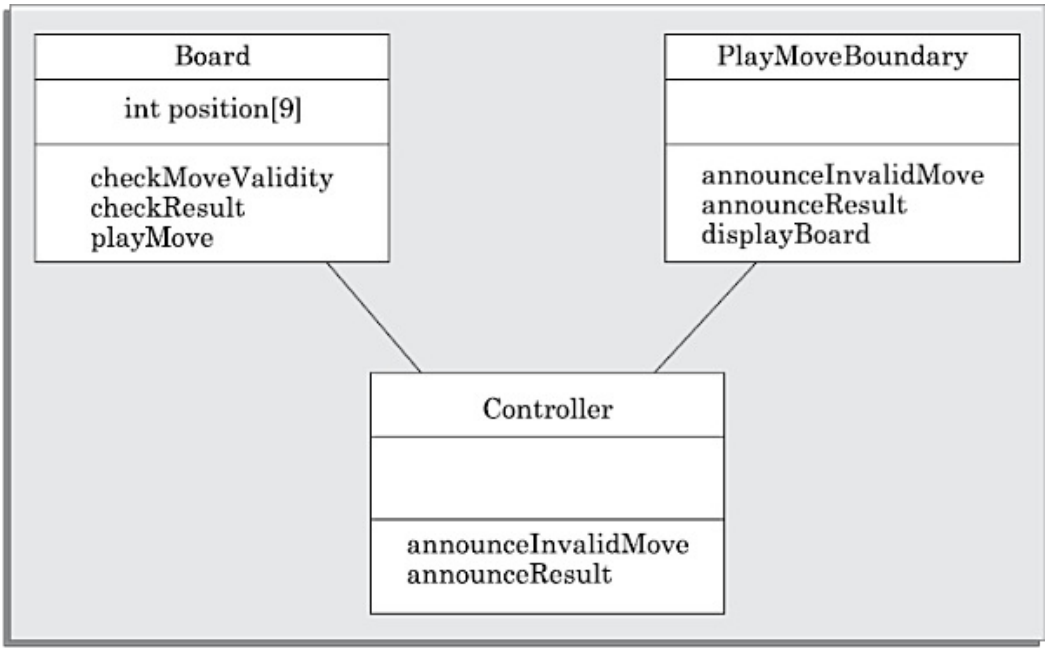

Figure 8.15: Class diagram for Example 8.2.

Example 8.3 Consider the Supermarket prizes scheme software discussed in Example 6.3. The step-by-step analysis and design workout of this problem is as follows:

- The use case model is shown in Figure 8.16.

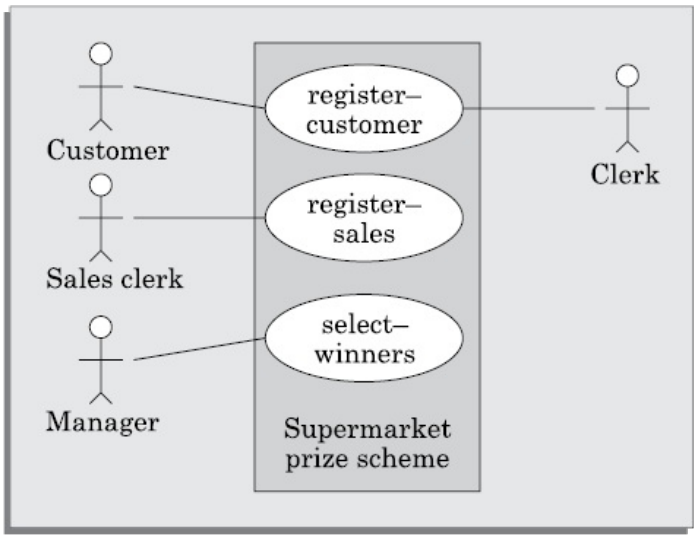

Figure 8.16: Use case model for Example 8.3.

- The initial domain model is shown in Figure 8.17 (a).
- The domain model after adding the boundary and control classes is shown in Figure 8.17 (b).

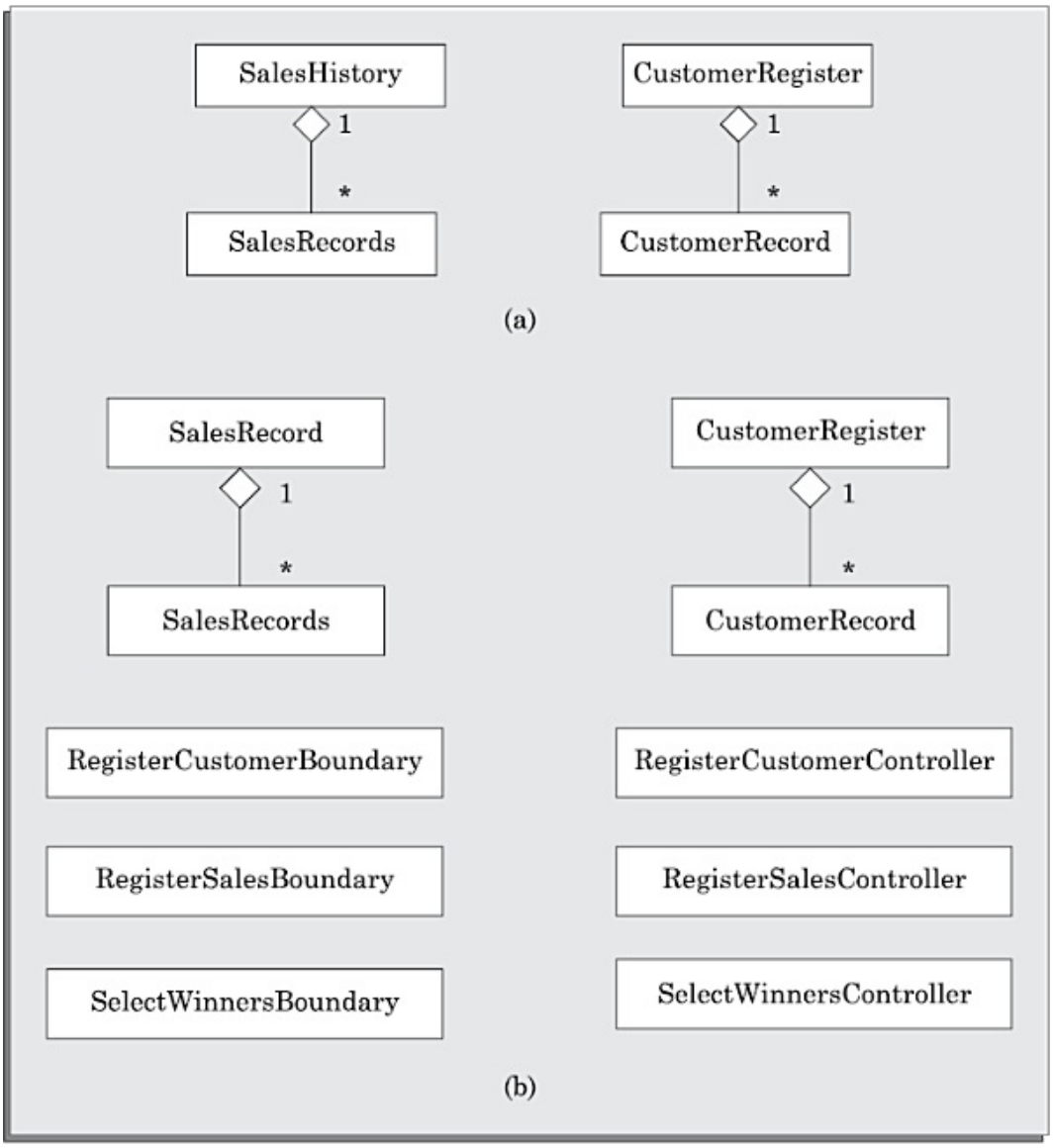

Figure 8.17: (a) Initial domain model (b) Refined domain model for Example 8.3.

- Sequence diagram for the select winner list use case Figure 8.18.

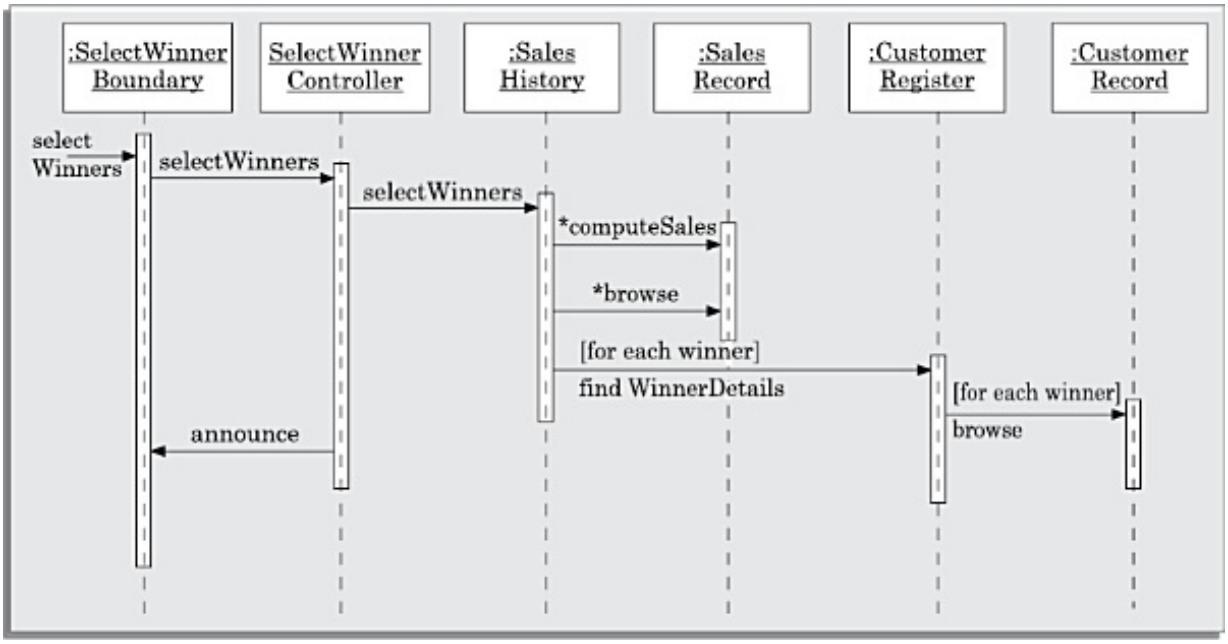

Figure 8.18: Sequence diagram for the select winner list use case of Example 8.3.

- Sequence diagram for the register use case is shown in Figure 8.19.

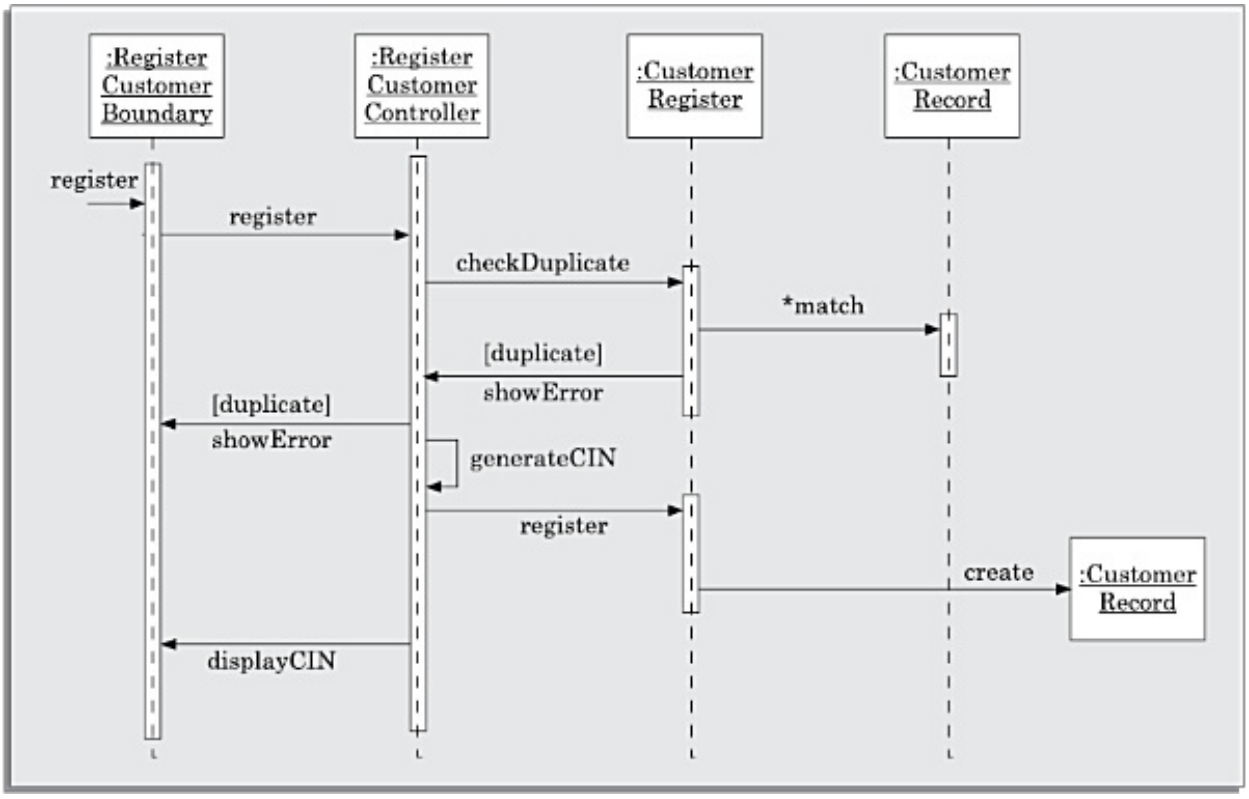

Figure 8.19: Sequence diagram for the register customer use case of Example 8.3.

- Sequence diagram for the register sales use case Figure 8.20. In this use case, since the responsibility of the `RegisterSalesController` is trivial, the controller class can be deleted and the sequence diagram

has been redrawn in Figure 8.21 after incorporating this change.

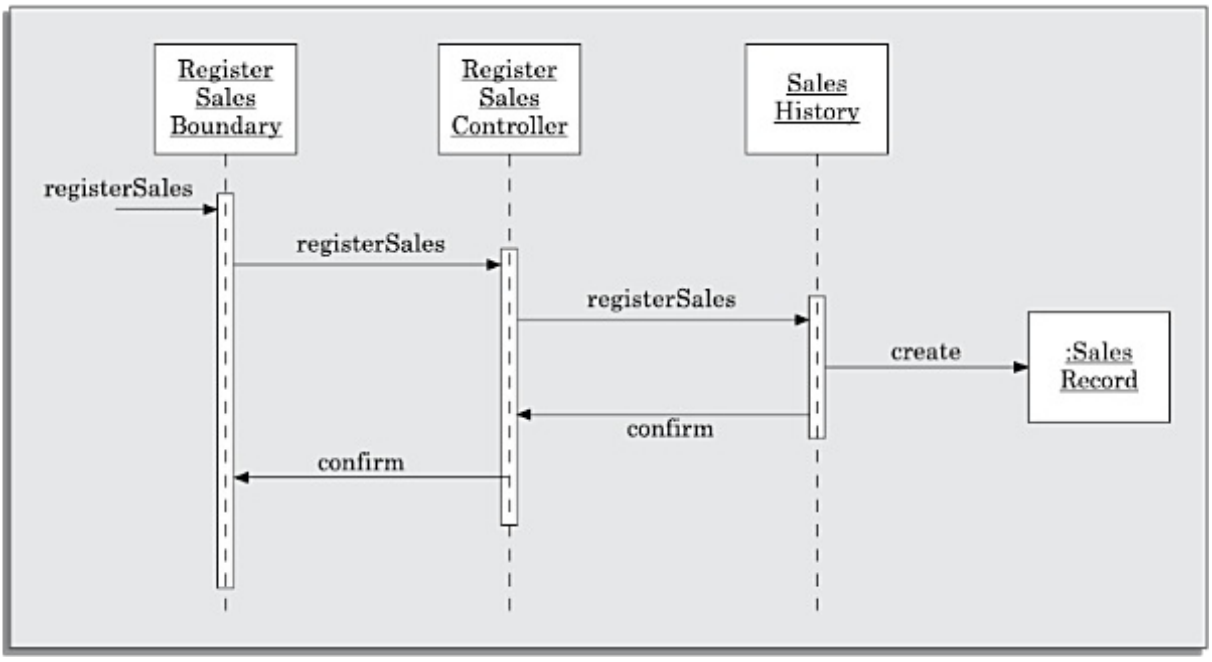

Figure 8.20: Sequence diagram for the register sales use case of Example 8.3.

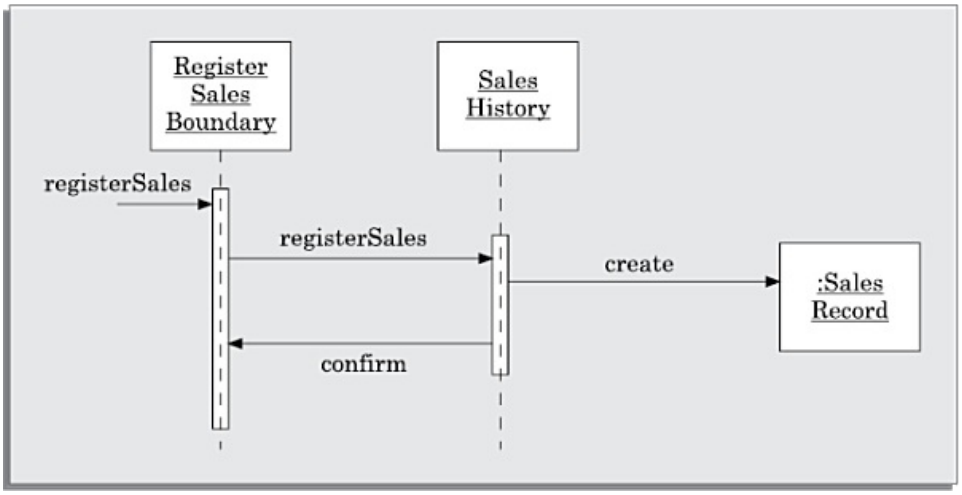

Figure 8.21: Refined sequence diagram for the register sales use case of Example 8.3.

- Class diagram is shown in Figure 8.22. Observe that the messages of the sequence diagrams (Figs. 8.18 to 8.20) of the different use cases have been populated as the methods of the corresponding classes.

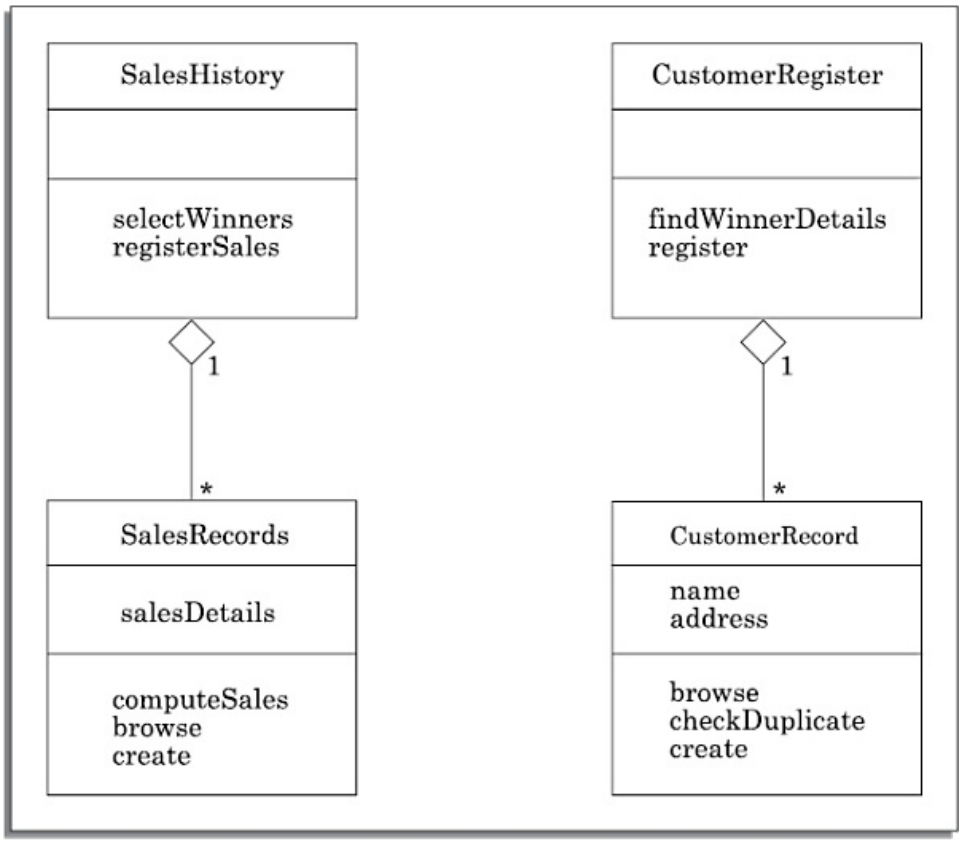

Figure 8.22: Class diagram for Example 8.3.

8.5 OOD GOODNESS CRITERIA

We have seen that several subjective judgments are made while arriving at an object-oriented design (OOD) solution. Depending on the exact judgment, several alternative design solutions to the same problem are possible. In order to be able to determine which of several alternative design solutions is better, we need to identify some criteria to determine which of two alternative designs is preferable. The following are some of the accepted criteria for judging the goodness of a design:

Coupling: Excessive coupling between objects is detrimental to modular design and prevents reuse. The number of messages between two objects or among a group of objects should be minimum. Increase in the number of message exchanges between two objects results in increasing the coupling between them.

Cohesion: Cohesion in the design should be high. In OOD, we are concerned about cohesion at three levels:

Cohesiveness of the individual methods: Cohesiveness of each individual method in a class is desirable. This requires that each method should perform only a certain well-defined function. This is desirable since it assures that the

methods of an object do actions for which the object is naturally responsible, i.e., it assures that no action has been improperly mapped to an object.

Cohesiveness of the data and methods within a class: The classes in a hierarchy should be coherent. For example, from the base class Issuable in a library, only issuable items should be derived. It would be highly inappropriate to derive a LibraryMember class from the Issuable class.

Cohesiveness of an entire class hierarchy: Cohesiveness of methods within a class is desirable since it promotes encapsulation of the objects.

Hierarchy and factoring guidelines: A base class should not have too many subclasses. If too many subclasses are derived from a single base class, then it becomes difficult to understand the design. In fact, there should approximately be no more than 7 ± 2 classes derived from a base class at any level.

Keeping message protocols simple Complex message protocols are an indication of excessive coupling among objects. If a message requires more than 3 parameters, then it is an indication of inferior design.

Number of methods: Classes having a large number of methods are likely to be more application-specific and also difficult to comprehend—limiting the possibility of their reuse. Therefore, classes should not have too many methods. In fact, the number of methods can be used as a measure of the complexity of a class. Maintaining and debugging classes having more than about seven methods can be problematic.

Depth of the inheritance tree: The deeper a class is in the class inheritance hierarchy, the greater is the number of methods it is likely to inherit. The height of the inheritance tree therefore should not be very large. Usually, the complexity of classes in a class hierarchy increase from the root (base) class towards the leaf classes in the class hierarchy.

Number of messages per use case: The number of messages per use case can be high, when either few objects interact by generating too many messages or large number of objects interact during the execution of the use case. If the methods of a large number of objects are invoked in a chain action in response to a single message, testing and debugging becomes complicated. If the execution of a use case shows a failure, detecting the error (bug) would be very difficult since the error can potentially exist in any of the participating objects. Further, there could be too much overhead incurred due to message passing and different classes possibly perform too little work. Each object essentially "passes the buck," rather than doing any

meaningful piece of work. Therefore, a single use case should not result in excessive message generation and transmission in a system.

Response for a class (RFC): The response for a class (RFC) is defined as the maximum number of methods and constructors of other objects that an instance of this class invokes. Please note that if the same method is called more than once, then it is counted only once. A class which calls more than about seven different methods is susceptible to errors.

SUMMARY

- Patterns are reusable solutions to recurring problems. We can spot a pattern in a problem if we are familiar with it. We discussed a few well-known patterns.
- We discussed a generic object-oriented analysis and design process. This analysis and design process requires first constructing the use case and domain models and then iterating through the interaction diagram to obtain the final class diagram.
- Even though we described a step-by-step methodology for object-oriented analysis and design; a good design can only be obtained through several iterations, trying out several alternative solutions. Practice of solving a large number of problems is particularly useful in arriving at a good design to a problem.
- We discussed the characteristics of a good design solution and defined some metrics that can be used to judge a design solution.

EXERCISES

1. Choose the correct option:
   (a) When using informal (natural language) description of a programming problem, which parts of the description will represent objects?
   (i) All of the nouns and some of the verbs.
   (ii) All of the verbs and some of the nouns.
   (iii) Some of the nouns.
   (iv) Some of the verbs
   (b) Which of the following parameters for a class correlates positively with the quality of the design solution:
   (i) Response for a class (RFC)

(ii) Depth of inheritance tree (DIT)
(iii) Number of messages per use case
(iv) Cohesiveness of data ad methods in a class

(c) Which of the following is a characteristic of a good object-oriented design:
(i) Deep class hierarchy
(ii) Large number of methods per class
(iii) Large number of message exchanges per use case
(iv) Moderate number of methods per class

2. What is the difference between object oriented analysis (OOA) and object oriented design (OOD)?
3. What is a pattern in the context of software development? Explain why patterns are considered to be an effective form of software reuse. What are the limitations of this approach for software reuse?
4. Which problem does the facade design pattern attempt to solve? What solutions does it offer?
5. Do you need to develop all the views of a system using all the modeling diagrams supported by UML? Justify your answer.
6. What are design patterns? What are the advantages of using design patterns? Name some popular design patterns.
7. What is unified process? What are the different phases of the unified process? What activities are carried out during each phase of the unified process?
8. Algorithms such as sorting are famed as reusable solutions to problems. Since patterns provide reusable solutions, is it correct to say that patterns are essentially algorithms? Explain your answer.
9. Why is "push from below" model of interaction between a GUI object with either a controller or domain object is not a good idea? What solution does MVC pattern offer in this regard?
10. State whether the following statements are **TRUE** o r **FALSE**. Give reasons behind your answers.
(a) Deep class hierarchies are signs of an object-oriented design done well.
(b) A large number of message exchanges among objects during the realisation of a use case indicates effective delegation of responsibilities and is a sign of good design.
(c) The difficulty of understanding a class in a class hierarchy increases

from the root to the leaves in a class hierarchy.

11. Define the term cohesion in the context of object-oriented software design.
12. What are some of the important criteria based on which it would be possible to determine which of two object-oriented design solutions to a problem is better?
13. Explain the differences between an architectural pattern, a design pattern, and an idiom in the context of object-oriented software development.
14. What is an anti-pattern? How antipatterns are helpful in arriving at a good design solution to a problem?
15. Briefly outline the important steps involved in developing a software system using a popular object-oriented design methodology.
16. Perform the object-oriented design for the development of the **Hotel automation software** (problem number 6.13) of Chapter 6.
17. Perform the object-oriented design for the development of the **Road Repair and Tracking Software (RRTS)** (problem number 6.15) of Chapter 6.
18. Perform the object-oriented design for the development of the **Book shop Automation Software (BAS)** (problem number 6.14) of Chappter 6.
19. Perform the object-oriented design for the development of the **Library Information System (LIS) software** (problem number 6.18) of Chapter 6.
20. Perform object-oriented design for developing the following **simulation software:**

    A factory has a certain category of machines that require frequent adjustments and repair. Each category of machine fails uniformly after continuous operation and the failure profile of the different categories of machines is given by its mean time to failure (MTTF). A certain number of adjusters are employed to keep the machines running. A service manager co-ordinates the activities of the adjusters. The service manager maintains a queue of inoperative machines. If there are machines waiting to be repaired, the service manager assigns the machine at the front of the queue to the next available adjuster. Likewise, when some adjusters are not busy, the service manager maintains a queue of idle adjusters and assigns the adjuster at the front

of the queue to the next machine that breaks down.

At any given time, one of the two queues will be empty. Thus, the service manager needs to maintain only a single queue, which when it is not empty contains only machines or only adjusters. The factory management wishes to get as much as possible out of its machines and adjusters. It is therefore interested in machine utilisation—the percentage of time a machine is up and running and the adjuster utilisation—the percentage of time an adjuster is busy. The goal of our simulation is then to see how the average machine and adjuster utilisation depend on such factors as the number of machines, the number of adjusters, the reliability of the machines in terms of mean time to failure (MTTF), and the productivity of the adjusters.

21. Consider the following Elevator Control Problem.

A software system (Elevator Controller) must control a set of 4 elevators for a building with 10 floors. Each elevator contains a set of buttons, each corresponding to a desired floor. These are called floor request buttons, since they indicate a request to go to a specific floor. Each elevator as well has a current floor indicator above the door. Each floor has two buttons for requesting elevators called elevator request buttons because they request an elevator. The elevator controller will receive all the signals from the passengers and decide on the control actions to be fed to the elevators

Each floor has a sliding door for each shaft arranged so that two door halves meet in the center when closed. When the elevator arrives at the floor, the doors opens at the same time the door on the elevator opens. The floor does have both pressure and optical sensors to prevent closing when an obstacle is between the two doors halves. If an obstruction is detected by either sensor, the door shall open. The door shall automatically close after a timeout period of five second after the door opens. The detection of an obstruction shall restart the door closure time after an obstruction is removed. There is a speaker on each floor that announces the arrival of an elevator.

On each floor, there are two elevator request buttons, one for UP and one for DOWN. On each floor above each elevator door, there is an indicator specifying the current floor of the elevator and another indicator for its current direction. The system shall respond to an elevator request by sending the nearest elevator that is either idle or already going in the requested direction. If no elevators are currently

available, the request shall be pending until an elevator meets the previously mentioned criterion. Once pressed, the request buttons are backlit to indicate that a request is pending. Pressing an elevator request button when a request for that direction is already pending, shall have no effect. When an elevator arrives to handle the request (i.e., it is slated to go in the selected direction), then the door shall stop closing and the door closure timer shall be reset.

To enhance safety, a cable tension sensor monitors the tension on the cable controlling the elevator. In the event of a failure (measured tension falls below a critical value), then four external locking clamps connected to running tracks in the shaft stop the elevator and hold it in place.

(a) Develop the domain model.

(b) Develop state chart model for the classes possessing significant number of states and behaviour.

1 Recall that during structured analysis a DFD model is developed and during structured design the DFD model is converted to structure chart representation.

2 Dictionary meaning of facade: The frontal appearance of a building.

3 The dictionary meaning of concept is idea, thing, object.

Chapter 9

USER INTERFACE DESIGN

The user interface portion of a software product is responsible for all interactions with the user. Almost every software product has a user interface (can you think of a software product that does not have any user interface?). In the early days of computer, no software product had any user interface. The computers those days were batch systems and no interactions with the users were supported. Now, we know that things are very different—almost every software product is highly interactive. The user interface part of a software product is responsible for all interactions with the end-user. Consequently, the user interface part of any software product is of direct concern to the end-users. No wonder then that many users often judge a software product based on its user interface. Aesthetics apart, an interface that is difficult to use leads to higher levels of user errors and ultimately leads to user dissatisfaction. Users become particularly irritated when a system behaves in an unexpected ways, i.e., issued commands do not carry out actions according to the intuitive expectations of the user. Normally, when a user starts using a system, he builds a mental model of the system and expects the system behaviour to conform to it. For example, if a user action causes one type of system activity and response under some context, then the user would expect similar system activity and response to occur for similar user actions in similar contexts. Therefore, sufficient care and attention should be paid to the design of the user interface of any software product.

Systematic development of the user interface is also important from another consideration. Development of a good user interface usually takes significant portion of the total system development effort. For many interactive applications, as much as 50 per cent of the total development effort is spent on developing the user interface part. Unless the user interface

is designed and developed in a systematic manner, the total effort required to develop the interface will increase tremendously. Therefore, it is necessary to carefully study various concepts associated with user interface design and understand various systematic techniques available for the development of user interface.

In this chapter, we first discuss some common terminologies and concepts associated with development of user interfaces. Then, we classify the different types of interfaces commonly being used. We also provide some guidelines for designing good interfaces, and discuss some tools for development of graphical user interfaces (GUIs). Finally, we present a GUI development methodology.

9.1 CHARACTERISTICS OF A GOOD USER INTERFACE

Before we start discussing anything about how to develop user interfaces, it is important to identify the different characteristics that are usually desired of a good user interface. Unless we know what exactly is expected of a good user interface, we cannot possibly design one. In the following subsections, we identify a few important characteristics of a good user interface:

Speed of learning: A good user interface should be easy to learn. Speed of learning is hampered by complex syntax and semantics of the command issue procedures. A good user interface should not require its users to memorise commands. Neither should the user be asked to remember information from one screen to another while performing various tasks using the interface. Besides, the following three issues are crucial to enhance the speed of learning:

— **Use of metaphors**[1] **and intuitive command names:** Speed of learning an interface is greatly facilitated if these are based on some day-to-day real-life examples or some physical objects with which the users are familiar with. The abstractions of real-life objects or concepts used in user interface design are called metaphors. If the user interface of a text editor uses concepts similar to the tools used by a writer for text editing such as cutting lines and paragraphs and pasting it at other places, users can immediately relate to it. Another popular metaphor is a shopping cart. Everyone knows how a shopping cart is used to make choices while purchasing items in a supermarket. If a user interface uses the shopping cart metaphor for designing the interaction style for a situation where

similar types of choices have to be made, then the users can easily understand and learn to use the interface. Also, learning is facilitated by intuitive command names and symbolic command issue procedures.

— **Consistency:** Once, a user learns about a command, he should be able to use the similar commands in different circumstances for carrying out similar actions. This makes it easier to learn the interface since the user can extend his knowledge about one part of the interface to the other parts. Thus, the different commands supported by an interface should be consistent.

— **Component-based interface:** Users can learn an interface faster if the interaction style of the interface is very similar to the interface of other applications with which the user is already familiar with. This can be achieved if the interfaces of different applications are developed using some standard user interface components. This, in fact, is the theme of the component-based user interface discussed in Section 9.5.

The speed of learning characteristic of a user interface can be determined by measuring the training time and practice that users require before they can effectively use the software.

Speed of use: Speed of use of a user interface is determined by the time and user effort necessary to initiate and execute different commands. This characteristic of the interface is some times referred to as productivity support of the interface. It indicates how fast the users can perform their intended tasks. The time and user effort necessary to initiate and execute different commands should be minimal. This can be achieved through careful design of the interface. For example, an interface that requires users to type in lengthy commands or involves mouse movements to different areas of the screen that are wide apart for issuing commands can slow down the operating speed of users. The most frequently used commands should have the smallest length or be available at the top of a menu to minimise the mouse movements necessary to issue commands.

Speed of recall: Once users learn how to use an interface, the speed with which they can recall the command issue procedure should be maximised. This characteristic is very important for intermittent users. Speed of recall is improved if the interface is based on some metaphors, symbolic command issue procedures, and intuitive command names.

Error prevention: A good user interface should minimise the scope of committing errors while initiating different commands. The error rate of an interface can be easily determined by monitoring the errors committed by an

average users while using the interface. This monitoring can be automated by instrumenting the user interface code with monitoring code which can record the frequency and types of user error and later display the statistics of various kinds of errors committed by different users. Consistency of names, issue procedures, and behaviour of similar commands and the simplicity of the command issue procedures minimise error possibilities. Also, the interface should prevent the user from entering wrong values.

Aesthetic and attractive: A good user interface should be attractive to use. An attractive user interface catches user attention and fancy. In this respect, graphics-based user interfaces have a definite advantage over text-based interfaces.

Consistency: The commands supported by a user interface should be consistent. The basic purpose of consistency is to allow users to generalise the knowledge about aspects of the interface from one part to another. Thus, consistency facilitates speed of learning, speed of recall, and also helps in reduction of error rate

Feedback: A good user interface must provide feedback to various user actions. Especially, if any user request takes more than few seconds to process, the user should be informed about the state of the processing of his request. In the absence of any response from the computer for a long time, a novice user might even start recovery/shutdown procedures in panic. If required, the user should be periodically informed about the progress made in processing his command.

Support for multiple skill levels: A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users. This is necessary because users with different levels of experience in using an application prefer different types of user interfaces. Experienced users are more concerned about the efficiency of the command issue procedure, whereas novice users pay importance to usability aspects. Very cryptic and complex commands discourage a novice, whereas elaborate command sequences make the command issue procedure very slow and therefore put off experienced users. When someone uses an application for the first time, his primary concern is speed of learning. After using an application for extended periods of time, he becomes familiar with the operation of the software. As a user becomes more and more familiar with an interface, his focus shifts from usability aspects to speed of command issue aspects. Experienced users look for options such as “hot-keys”, “macros”, etc.

Thus, the skill level of users improves as they keep using a software product and they look for commands to suit their skill levels.

Error recovery (undo facility): While issuing commands, even the expert users can commit errors. Therefore, a good user interface should allow a user to undo a mistake committed by him while using the interface. Users are inconvenienced if they cannot recover from the errors they commit while using a software. If the users cannot recover even from very simple types of errors, they feel irritated, helpless, and out of control.

User guidance and on-line help: Users seek guidance and on-line help when they either forget a command or are unaware of some features of the software. Whenever users need guidance or seek help from the system, they should be provided with appropriate guidance and help.

9.2 BASIC CONCEPTS

In this section, we first discuss some basic concepts in user guidance and on-line help system. Next, we examine the concept of a mode-based and a modeless interface and the advantages of a graphical interface.

9.2.1 User Guidance and On-line Help

Users may seek help about the operation of the software any time while using the software. This is provided by the on-line help system. This is different from the guidance and error messages which are flashed automatically without the user asking for them. The guidance messages prompt the user regarding the options he has regarding the next command, and the status of the last command, etc.

On-line help system: Users expect the on-line help messages to be tailored to the context in which they invoke the "help system". Therefore, a good on-line help system should keep track of what a user is doing while invoking the help system and provide the output message in a context-dependent way. Also, the help messages should be tailored to the user's experience level. Further, a good on-line help system should take advantage of any graphics and animation characteristics of the screen and should not just be a copy of the user's manual.

Guidance messages: The guidance messages should be carefully designed to prompt the user about the next actions he might pursue, the current status of the system, the progress so far made in processing his last command, etc. A good guidance system should have different levels of sophistication for

different categories of users. For example, a user using a command language interface might need a different type of guidance compared to a user using a menu or iconic interface (These different types of interfaces are discussed later in this chapter). Also, users should have an option to turn off the detailed messages.

Error messages: Error messages are generated by a system either when the user commits some error or when some errors encountered by the system during processing due to some exceptional conditions, such as out of memory, communication link broken, etc. Users do not like error messages that are either ambiguous or too general such as "invalid input or system error". Error messages should be polite. Error messages should not have associated noise which might embarrass the user. The message should suggest how a given error can be rectified. If appropriate, the user should be given the option of invoking the on-line help system to find out more about the error situation.

9.2.2 Mode-based versus Modeless Interface

A mode is a state or collection of states in which only a subset of all user interaction tasks can be performed. In a modeless interface, the same set of commands can be invoked at any time during the running of the software. Thus, a modeless interface has only a single mode and all the commands are available all the time during the operation of the software. On the other hand, in a mode-based interface, different sets of commands can be invoked depending on the mode in which the system is, i.e., the mode at any instant is determined by the sequence of commands already issued by the user.

A mode-based interface can be represented using a state transition diagram, where each node of the state transition diagram would represent a mode. Each state of the state transition diagram can be annotated with the commands that are meaningful in that state.

9.2.3 Graphical User Interface (GUI) versus Text-based User Interface

Let us compare various characteristics of a GUI with those of a text-based user interface:

- In a GUI multiple windows with different information can simultaneously be displayed on the user screen. This is perhaps one of

the biggest advantages of GUI over text- based interfaces since the user has the flexibility to simultaneously interact with several related items at any time and can have access to different system information displayed in different windows.

- Iconic information representation and symbolic information manipulation is possible in a GUI. Symbolic information manipulation such as dragging an icon representing a file to a trash for deleting is intuitively very appealing and the user can instantly remember it.
- A GUI usually supports command selection using an attractive and user-friendly menu selection system.
- In a GUI, a pointing device such as a mouse or a light pen can be used for issuing commands. The use of a pointing device increases the efficacy of command issue procedure.
- On the flip side, a GUI requires special terminals with graphics capabilities for running and also requires special input devices such a mouse. On the other hand, a text-based user interface can be implemented even on a cheap alphanumeric display terminal. Graphics terminals are usually much more expensive than alphanumeric terminals. However, display terminals with graphics capability with bit-mapped high-resolution displays and significant amount of local processing power have become affordable and over the years have replaced text-based terminals on all desktops. Therefore, the emphasis of this chapter is on GUI design rather than text-based user interface design.

9.3 TYPES OF USER INTERFACES

Broadly speaking, user interfaces can be classified into the following three categories:

- Command language-based interfaces
- Menu-based interfaces
- Direct manipulation interfaces

Each of these categories of interfaces has its own characteristic advantages and disadvantages. Therefore, most modern applications use a careful combination of all these three types of user interfaces for implementing the user command repertoire. It is very difficult to come up with a simple set of

guidelines as to which parts of the interface should be implemented using what type of interface. This choice is to a large extent dependent on the experience and discretion of the designer of the interface. However, a study of the basic characteristics and the relative advantages of different types of interfaces would give a fair idea to the designer regarding which commands should be supported using what type of interface. In the following three subsections, we briefly discuss some important characteristics, advantages, and disadvantages of using each type of user interface.

9.3.1 Command Language-based Interface

A command language-based interface—as the name itself suggests, is based on designing a command language which the user can use to issue the commands. The user is expected to frame the appropriate commands in the language and type them appropriately whenever required. A simple command language-based interface might simply assign unique names to the different commands. However, a more sophisticated command language-based interface may allow users to compose complex commands by using a set of primitive commands. Such a facility to compose commands dramatically reduces the number of command names one would have to remember. Thus, a command language-based interface can be made concise requiring minimal typing by the user. Command language-based interfaces allow fast interaction with the computer and simplify the input of complex commands.

Among the three categories of interfaces, the command language interface allows for most efficient command issue procedure requiring minimal typing. Further, a command language-based interface can be implemented even on cheap alphanumeric terminals. Also, a command language-based interface is easier to develop compared to a menu-based or a direct-manipulation interface because compiler writing techniques are well developed. One can systematically develop a command language interface by using the standard compiler writing tools Lex and Yacc.

However, command language-based interfaces suffer from several drawbacks. Usually, command language-based interfaces are difficult to learn and require the user to memorise the set of primitive commands. Also, most users make errors while formulating commands in the command language and also while typing them. Further, in a command language-based interface, all interactions with the system is through a key-board and cannot take advantage of effective interaction devices such as a mouse. Obviously, for

casual and inexperienced users, command language-based interfaces are not suitable.

Issues in designing a command language-based interface

Two overbearing command design issues are to reduce the number of primitive commands that a user has to remember and to minimise the total typing required. We elaborate these considerations in the following:

- The designer has to decide what mnemonics (command names) to use for the different commands. The designer should try to develop meaningful mnemonics and yet be concise to minimise the amount of typing required. For example, the shortest mnemonic should be assigned to the most frequently used commands.
- The designer has to decide whether the users will be allowed to redefine the command names to suit their own preferences. Letting a user define his own mnemonics for various commands is a useful feature, but it increases the complexity of user interface development.
- The designer has to decide whether it should be possible to compose primitive commands to form more complex commands. A sophisticated command composition facility would require the syntax and semantics of the various command composition options to be clearly and unambiguously specified. The ability to combine commands is a powerful facility in the hands of experienced users, but quite unnecessary for inexperienced users.

9.3.2 Menu-based Interface

An important advantage of a menu-based interface over a command language-based interface is that a menu-based interface does not require the users to remember the exact syntax of the commands. A menu-based interface is based on recognition of the command names, rather than recollection. Humans are much better in recognising something than recollecting it. Further, in a menu-based interface the typing effort is minimal as most interactions are carried out through menu selections using a pointing device. This factor is an important consideration for the occasional user who cannot type fast.

However, experienced users find a menu-based user interface to be slower than a command language-based interface because an experienced user can

type fast and can get speed advantage by composing different primitive commands to express complex commands. Composing commands in a menu-based interface is not possible. This is because of the fact that actions involving logical connectives (and, or, etc.) are awkward to specify in a menu-based system. Also, if the number of choices is large, it is difficult to design a menu-based interfae. A moderate-sized software might need hundreds or thousands of different menu choices. In fact, a major challenge in the design of a menu-based interface is to structure large number of menu choices into manageable forms. In the following, we discuss some of the techniques available to structure a large number of menu items:

Scrolling menu: Sometimes the full choice list is large and cannot be displayed within the menu area, scrolling of the menu items is required. This would enable the user to view and select the menu items that cannot be accommodated on the screen. However, in a scrolling menu all the commands should be highly correlated, so that the user can easily locate a command that he needs. This is important since the user cannot see all the commands at any one time. An example situation where a scrolling menu is frequently used is font size selection in a document processor (see Figure 9.1). Here, the user knows that the command list contains only the font sizes that are arranged in some order and he can scroll up or down to find the size he is looking for. However, if the commands do not have any definite ordering relation, then the user would have to in the worst case, scroll through all the commands to find the exact command he is looking for, making this organisation inefficient.

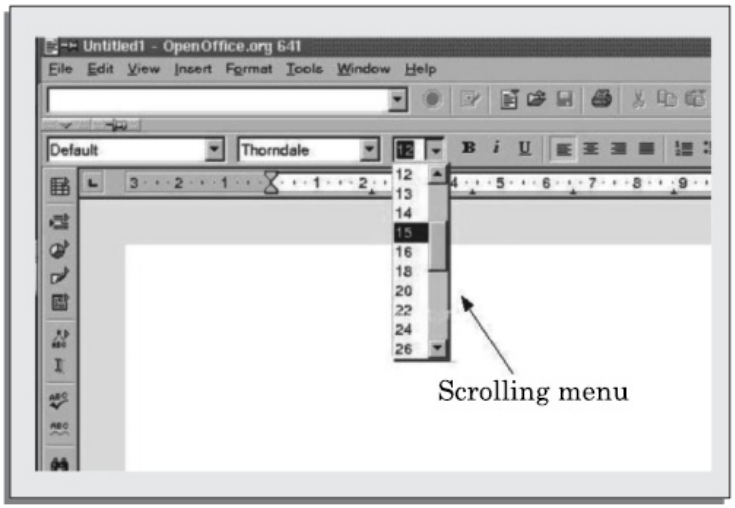

Figure 9.1: Font size selection using scrolling menu.

Walking menu: Walking menu is very commonly used to structure a large collection of menu items. In this technique, when a menu item is selected, it

causes further menu items to be displayed adjacent to it in a sub-menu. An example of a walking menu is shown in Figure 9.2. A walking menu can successfully be used to structure commands only if there are tens rather than hundreds of choices since each adjacently displayed menu does take up screen space and the total screen area is after all limited.

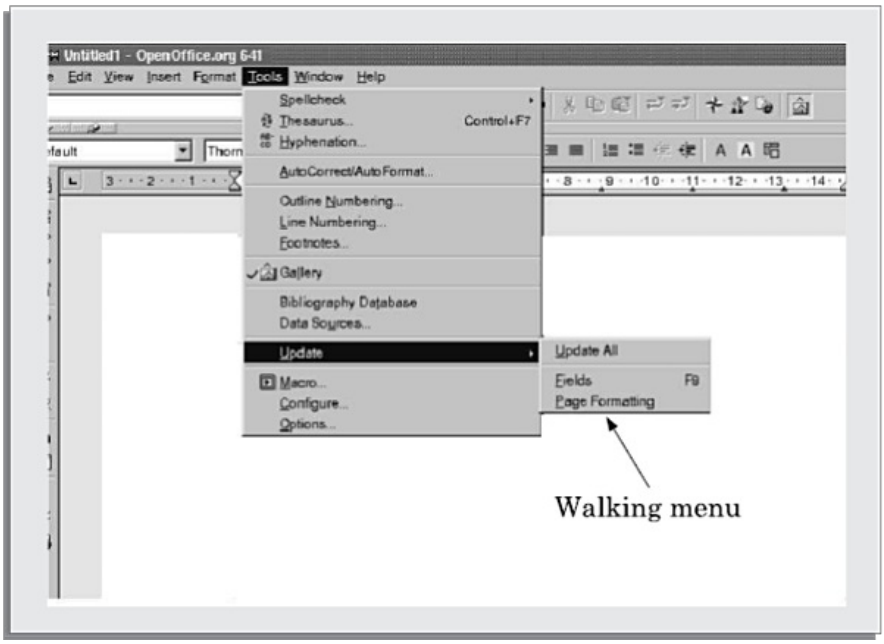

Figure 9.2: Example of walking menu.

Hierarchical menu: This type of menu is suitable for small screens with limited display area such as that in mobile phones. In a hierarchical menu, the menu items are organised in a hierarchy or tree structure. Selecting a menu item causes the current menu display to be replaced by an appropriate sub-menu. Thus in this case, one can consider the menu and its various sub-menu to form a hierarchical tree-like structure. Walking menu can be considered to be a form of hierarchical menu which is practicable when the tree is shallow. Hierarchical menu can be used to manage large number of choices, but the users are likely to face navigational problems because they might lose track of where they are in the menu tree. This probably is the main reason why this type of interface is very rarely used.

9.3.3 Direct Manipulation Interfaces

Direct manipulation interfaces present the interface to the user in the form of visual models (i.e., icons[2] or objects). For this reason, direct manipulation interfaces are sometimes called as iconic interfaces. In this type of interface, the user issues commands by performing actions on the visual representations of the objects, e.g., pull an icon

representing a file into an icon representing a trash box, for deleting the file.

Important advantages of iconic interfaces include the fact that the icons can be recognised by the users very easily, and that icons are language-independent. However, experienced users find direct manipulation interfaces very for too. Also, it is difficult to give complex commands using a direct manipulation interface. For example, if one has to drag an icon representing the file to a trash box icon for deleting a file, then in order to delete all the files in the directory one has to perform this operation individually for all files —which could be very easily done by issuing a command like delete *.*.

9.4 FUNDAMENTALS OF COMPONENT-BASED GUI DEVELOPMENT

Graphical user interfaces became popular in the 1980s. The main reason why there were very few GUI-based applications prior to the eighties is that graphics terminals were too expensive. For example, the price of a graphics terminal those days was much more than what a high-end personal computer costs these days. Also, the graphics terminals were of storage tube type and lacked raster capability.

One of the first computers to support GUI-based applications was the Apple Macintosh computer. In fact, the popularity of the Apple Macintosh computer in the early eighties is directly attributable to its GUI. In those early days of GUI design, the user interface programmer typically started his interface development from the scratch. He would starting from simple pixel display routines, write programs to draw lines, circles, text, etc. He would then develop his own routines to display menu items, make menu choices, etc. The current user interface style has undergone a sea change compared to the early style.

The current style of user interface development is component-based. It recognises that every user interface can easily be built from a handfuls of predefined components such as menus, dialog boxes, forms, etc. Besides the standard components, and the facilities to create good interfaces from them, one of the basic support available to the user interface developers is the window system. The window system lets the application programmer create and manipulate windows without having to write the basic windowing functions.

In the following subsections, we provide an overview of the window management system, the component-based development style, and visual

programming.

9.4.1 Window System

Most modern graphical user interfaces are developed using some window system. A window system can generate displays through a set of windows. Since a window is the basic entity in such a graphical user interface, we need to first discuss what exactly a window is.

Window: A window is a rectangular area on the screen. A window can be considered to be a virtual screen, in the sense that it provides an interface to the user for carrying out independent activities, e.g., one window can be used for editing a program and another for drawing pictures, etc.

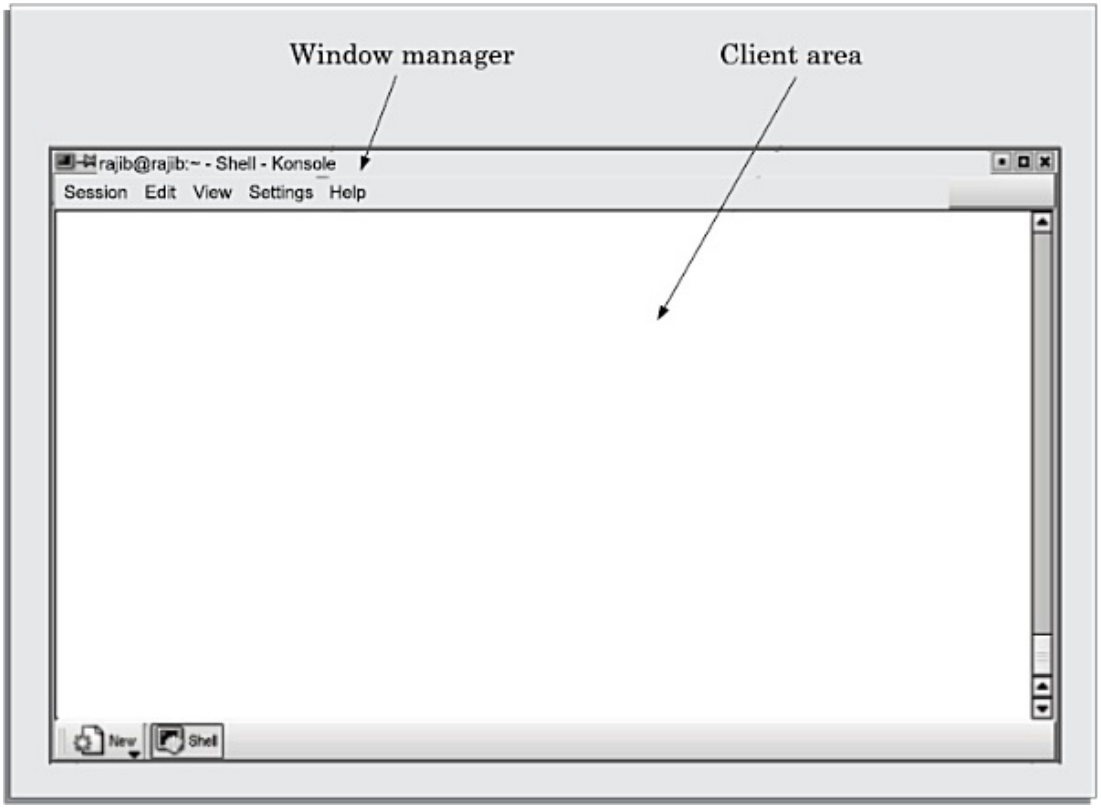

Figure 9.3: Window with client and user areas marked.

A window can be divided into two parts—client part, and non-client part. The client area makes up the whole of the window, except for the borders and scroll bars. The client area is the area available to a client application for display. The non-client-part of the window determines the look and feel of the window. The look and feel defines a basic behaviour for all windows, such as creating, moving, resizing, iconifying of the windows. The window manager is responsible for managing and maintaining the non-client area of a window. A basic window with its different parts is shown in Figure 9.3.

Window management system (WMS)

A graphical user interface typically consists of a large number of windows. Therefore, it is necessary to have some systematic way to manage these windows. Most graphical user interface development environments do this through a window management system (WMS). A window management system is primarily a resource manager. It keeps track of the screen area resource and allocates it to the different windows that seek to use the screen. From a broader perspective, a WMS can be considered as a user interface management system (UIMS)—which not only does resource management, but also provides the basic behaviour to the windows and provides several utility routines to the application programmer for user interface development. A WMS simplifies the task of a GUI designer to a great extent by providing the basic behaviour to the various windows such as move, resize, iconify, etc. as soon as they are created and by providing the basic routines to manipulate the windows from the application program such as creating, destroying, changing different attributes of the windows, and drawing text, lines, etc.

A WMS consists of two parts (see Figure 9.4):

- a window manager, and
- a window system.

These components of the WMS are discussed in the following subsection.

Window manager and window system: The window manager is built on the top of the window system in the sense that it makes use of various services provided by the window system. The window manager and not the window system determines how the windows look and behave. In fact, several kinds of window managers can be developed based on the same window system. The window manager can be considered as a special kind of client that makes use of the services (function calls) supported by the window system. The application programmer can also directly invoke the services of the window system to develop the user interface. The relationship between the window manager, window system, and the application program is shown in Figure 9.4. This figure shows that the end-user can either interact with the application itself or with the window manager (resize, move, etc.) and both the application and the window manger invoke services of the window manager.

Window manager is the component of WMS with which the end user interacts to do

various window-related operations such as window repositioning, window resizing, iconification, etc.

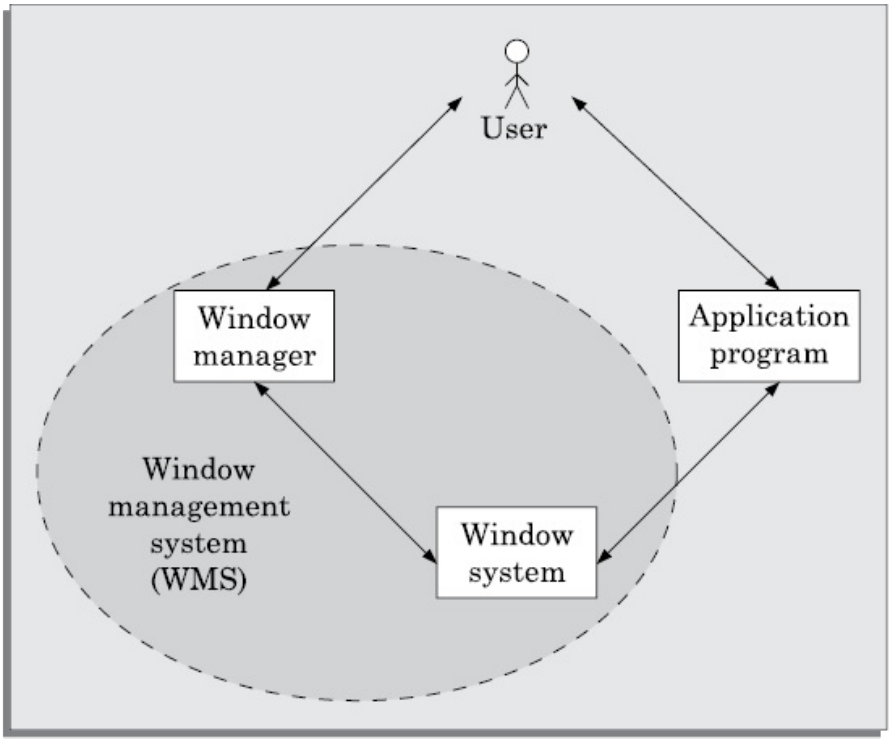

Figure 9.4: Window management system.

It is usually cumbersome to develop user interfaces using the large set of routines provided by the basic window system. Therefore, most user interface development systems usually provide a high-level abstraction called widgets for user interface development. A widget is the short form of a window object. We know that an object is essentially a collection of related data with several operations defined on these data which are available externally to operate on these data. The data of an window object are the geometric attributes (such as size, location etc.) and other attributes such as its background and foreground colour, etc. The operations that are defined on these data include, resize, move, draw, etc.

Widgets are the standard user interface components. A user interface is usually made up by integrating several widgets. A few important types of widgets normally provided with a user interface development system are described in Section 9.4.2.

Component-based development

A development style based on widgets is called component-based (or widget-based) GUI development style. There are several important advantages of using a widget-based design style. One of the most

important reasons to use widgets as building blocks is because they help users learn an interface fast. In this style of development, the user interfaces for different applications are built from the same basic components. Therefore, the user can extend his knowledge of the behaviour of the standard components from one application to the other. Also, the component-based user interface development style reduces the application programmer's work significantly as he is more of a user interface component integrator than a programmer in the traditional sense. In the following section, we will discuss some of these popular widgets.

Visual programming

Visual programming is the drag and drop style of program development. In this style of user interface development, a number of visual objects (icons) representing the GUI components are provided by the programming environment. The application programmer can easily develop the user interface by dragging the required component types (e.g., menu, forms, etc.) from the displayed icons and placing them wherever required. Thus, visual programming can be considered as program development through manipulation of several visual objects. Reuse of program components in the form of visual objects is an important aspect of this style of programming. Though popular for user interface development, this style of programming can be used for other applications such as Computer-Aided Design application (e.g., factory design), simulation, etc. User interface development using a visual programming language greatly reduces the effort required to develop the interface.

Examples of popular visual programming languages are Visual Basic, Visual C++, etc. Visual C++ provides tools for building programs with window-based user interfaces for Microsoft Windows environments. In visual C++ you usually design menu bars, icons, and dialog boxes, etc. before adding them to your program. These objects are called as resources. You can design shape, location, type, and size of the dialog boxes before writing any C++ code for the application.

9.4.2 Types of Widgets

Different interface programming packages support different widget sets. However, a surprising number of them contain similar kinds of widgets,

so that one can think of a generic widget set which is applicable to most interfaces. The following widgets we have chosen as representatives of this generic class.

Label widget: This is probably one of the simplest widgets. A label widget does nothing except to display a label, i.e., it does not have any other interaction capabilities and is not sensitive to mouse clicks. A label widget is often used as a part of other widgets.

Container widget: These widgets do not stand by themselves, but exist merely to contain other widgets. Other widgets are created as children of the container widget. When the container widget is moved or resized, its children widget also get moved or resized. A container widget has no callback routines associated with it.

Pop-up menu: These are transient and task specific. A pop-up menu appears upon pressing the mouse button, irrespective of the mouse position.

Pull-down menu : These are more permanent and general. You have to move the cursor to a specific location and pull down this type of menu.

Dialog boxes: We often need to select multiple elements from a selection list. A dialog box remains visible until explicitly dismissed by the user. A dialog box can include areas for entering text as well as values. If an apply command is supported in a dialog box, the newly entered values can be tried without dismissing the box. Though most dialog boxes ask you to enter some information, there are some dialog boxes which are merely informative, alerting you to a problem with your system or an error you have made. Generally, these boxes ask you to read the information presented and then click OK to dismiss the box.

Push button: A push button contains key words or pictures that describe the action that is triggered when you activate the button. Usually, the action related to a push button occurs immediately when you click a push button unless it contains an ellipsis (. . .). A push button with an ellipsis generally indicates that another dialog box will appear.

Radio buttons: A set of radio buttons are used when only one option has to be selected out of many options. A radio button is a hollow circle followed by text describing the option it stands for. When a radio button is selected, it appears filled and the previously selected radio button from the group is unselected. Only one radio button from a group can be selected at any time. This operation is similar to that of the band selection buttons that were available in old radios.

Combo boxes: A combo box looks like a button until the user interacts with it. When the user presses or clicks it, the combo box displays a menu of items to choose from. Normally a combo box is used to display either one-of-many choices when space is limited, the number of choices is large, or when the menu items are computed at run-time.

9.4.3 An Overview of X-Window/MOTIF

One of the important reasons behind the extreme popularity of the X-window system is probably due to the fact that it allows development of portable GUIs. Applications developed using the X-window system are device-independent. Also, applications developed using the X-window system become network independent in the sense that the interface would work just as well on a terminal connected anywhere on the same network as the computer running the application is. Network-independent GUI operation has been schematically represented in Figure 9.5. Here, A is the computer application in which the application is running. B can be any computer on the network from where you can interact with the application. Network-independent GUI was pioneered by the X-window system in the mid-eighties at MIT (Massachusetts Institute of Technology) with support from DEC (Digital Equipment Corporation). Now-a-days many user interface development systems support network-independent GUI development, e.g., the AWT and Swing components of Java.

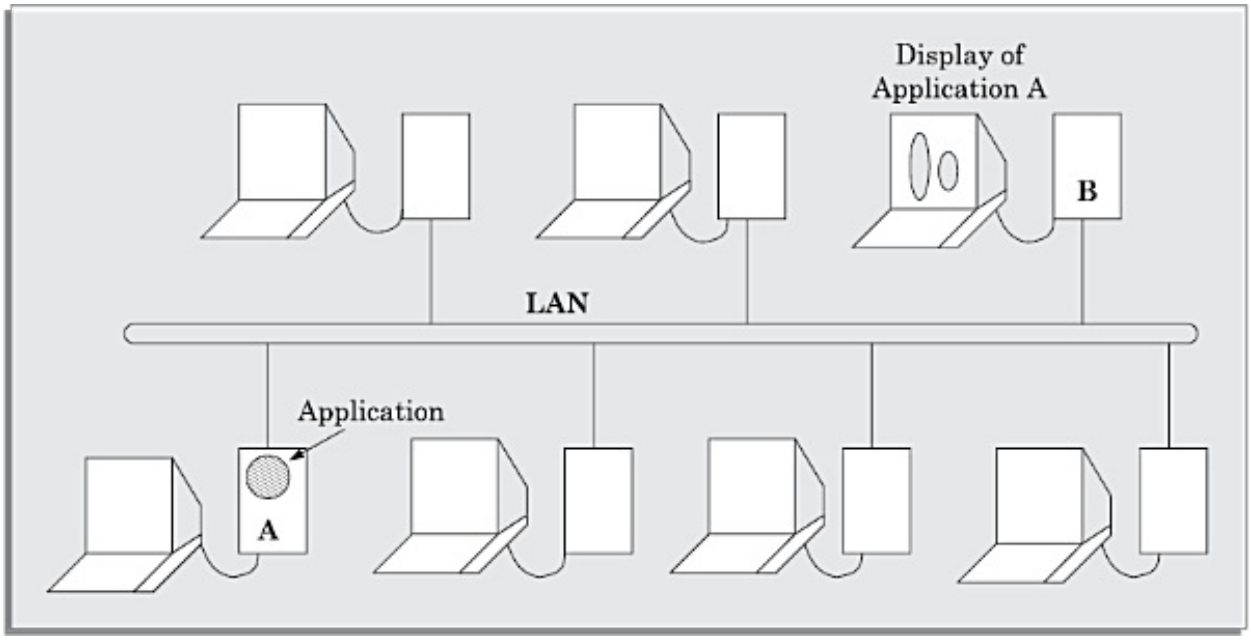

Figure 9.5: Network-independent GUI.

The X-window functions are low level functions written in C language which

can be called from application programs. But only the very serious application designer would program directly using the X-windows library routines. Built on top of X-windows are higher level functions collectively called Xtoolkit, which consists of a set of basic widgets and a set of routines to manipulate these widgets. One of the most widely used widget sets is X/Motif. Digital Equipment Corporation (DEC) used the basic X-window functions to develop its own look and feel for interface designs called DECWindows. In the following, we shall provide a very brief overview of the X-window system and its architecture and the interested reader is referred to Scheifler et al. [1988] for further study on graphical user interface development using X-windows and Motif.

9.4.4 X Architecture

The X architecture is pictorially depicted in Figure 9.6. The different terms used in this diagram are explained as follows:

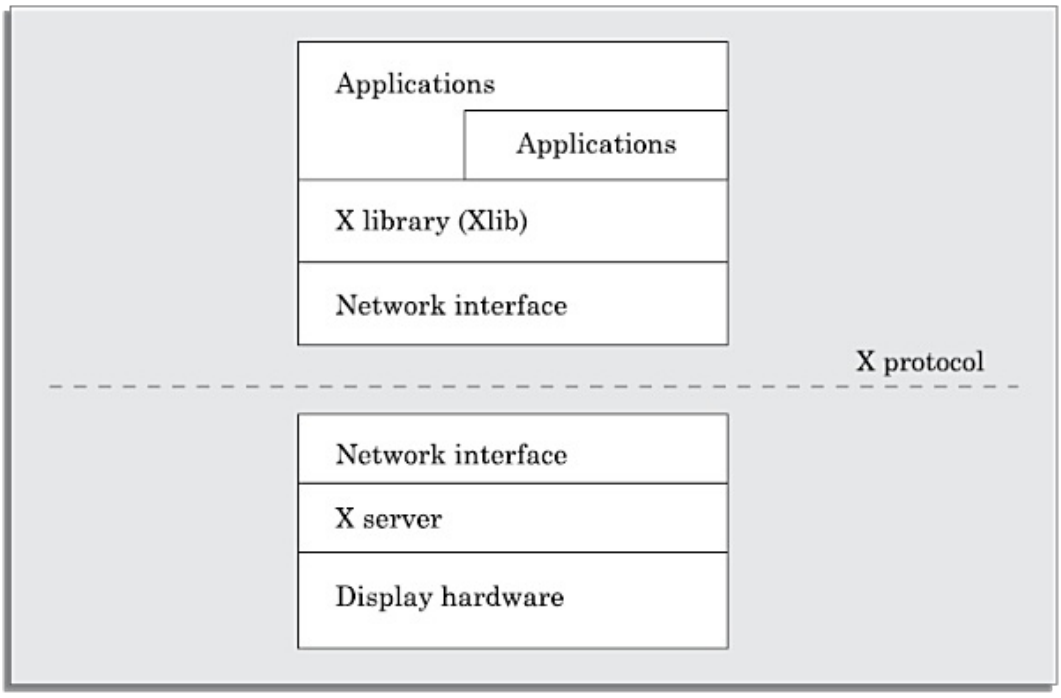

Figure 9.6: Architecture of the X System.

Xserver: The X server runs on the hardware to which the display and the key board are attached. The X server performs low-level graphics, manages window, and user input functions. The X server controls accesses to a bit-mapped graphics display resource and manages it.

X protocol. The X protocol defines the format of the requests between client applications and display servers over the network. The X protocol is designed to be independent of hardware, operating systems, underlying network

protocol, and the programming language used.

X library (Xlib). The Xlib provides a set of about 300 utility routines for applications to call. These routines convert procedure calls into requests that are transmitted to the server. Xlib provides low level primitives for developing an user interface, such as displaying a window, drawing characters and graphics on the window, waiting for specific events, etc.

Xtoolkit (Xt). The Xtoolkit consists of two parts: the intrinsics and the widgets. We have already seen that widgets are predefined user interface components such as scroll bars, menu bars, push buttons, etc. for designing GUIs. Intrinsics are a set of about a dozen library routines that allow a programmer to combine a set of widgets into a user interface. In order to develop a user interface, the designer has to put together the set of components (widgets) he needs, and then he needs to define the characteristics (called resources) and behaviour of these widgets by using the intrinsic routines to complete the development of the interface. Therefore, developing an interface using Xtoolkit is much easier than developing the same interface using only X library.

9.4.5 Size Measurement of a Component-based GUI

Lines of code (LOC) is not an appropriate metric to estimate and measure the size of a component-based GUI. This is because, the interface is developed by integrating several pre- built components. The different components making up an interface might have been in written using code of drastically different sizes. However, as far as the effort of the GUI developer who develops an interface by integrating the components may not be affected by the code size of the components he integrates.

A way to measure the size of a modern user interface is widget points (wp). The size of a user interface (in wp units) is simply the total number of widgets used in the interface. The size of an interface in wp units is a measure of the intricacy of the interface and is more or less independent of the implementation environment. The wp measure opens up chances for contracts on a measured amount of user interface functionality, instead of a vague definition of a complete system. However, till now there is no reported results to estimate the development effort in terms of the wp metric. An alternate way to compute the size of GUI is to simply count the number of screens. However, this would be inaccurate since a screen complexity can range from very simple to very complex.

9.5 A USER INTERFACE DESIGN METHODOLOGY

At present, no step-by-step methodology is available which can be followed by rote to come up with a good user interface. What we present in this section is a set of recommendations which you can use to complement your ingenuity. Even though almost all popular GUI design methodologies are user-centered, this concept has to be clearly distinguished from a user interface design by users. Before we start discussing about the user interface design methodology, let us distinguish between a user-centered design and a design by users.

- User-centered design is the theme of almost all modern user interface design techniques. However, user-centered design does not mean design by users. One should not get the users to design the interface, nor should one assume that the user's opinion of which design alternative is superior is always right. Though users may have good knowledge of the tasks they have to perrform using a GUI, but they may not know the GUI design issues.
- Users have good knowledge of the tasks they have to perform, they also know whether they find an interface easy to learn and use but they have less understanding and experience in GUI design than the GUI developers.

9.5.1 Implications of Human Cognition Capabilities on User Interface Design

An area of human-computer interaction where extensive research has been conducted is how human cognitive capabilities and limitations influence the way an interface should be designed. In the following subsections, we discuss some of the prominent issues that have been extensively reported in the literature.

Limited memory: Humans can remember at most seven unrelated items of information for short periods of time. Therefore, the GUI designer should not require the user to remember too many items of information at a time. It is the GUI designer's responsibility to anticipate what information the user will need at what point of each task and to ensure that the relevant information is displayed for the user to see. Showing the user some information at some point, and then asking him to recollect that information in a different screen where they no longer see the information, places a memory burden on the

user and should be avoided wherever possible.

Frequent task closure: Doing a task (except for very trivial tasks) requires doing several subtasks. When the system gives a clear feedback to the user that a task has been successfully completed, the user gets a sense of achievement and relief. The user can clear out information regarding the completed task from memory. This is known as task closure. When the overall task is fairly big and complex, it should be divided into subtasks, each of which has a clear subgoal which can be a closure point.

Recognition rather than recall. Information recall incurs a larger memory burden on the users and is to be avoided as far as possible. On the other hand, recognition of information from the alternatives shown to him is more acceptable.

Procedural versus ob ject-oriented: Procedural designs focus on tasks, prompting the user in each step of the task, giving them very few options for anything else. This approach is best applied in situations where the tasks are narrow and well-defined or where the users are inexperienced, such as a bank ATM. An object-oriented interface on the other hand focuses on objects. This allows the users a wide range of options.

9.5.2 A GUI Design Methodology

The GUI design methodology we present here is based on the seminal work of Frank Ludolph [Frank1998]. Our user interface design methodology consists of the following important steps:

- • Examine the use case model of the software. Interview, discuss, and review the GUI issues with the end-users.
- Task and object modelling.
- Metaphor selection.
- Interaction design and rough layout.
- Detailed presentation and graphics design.
- GUI construction.
- Usability evaluation.

Examining the use case model

We now elaborate the above steps in GUI design. The starting point for GUI design is the use case model. This captures the important tasks the users need to perform using the software. As far as possible, a user

interface should be developed using one or more metaphors. Metaphors help in interface development at lower effort and reduced costs for training the users. Over time, people have developed efficient methods of dealing with some commonly occurring situations. These solutions are the themes of the metaphors. Metaphors can also be based on physical objects such as a visitor's book, a catalog, a pen, a brush, a scissor, etc. A solution based on metaphors is easily understood by the users, reducing learning time and training costs. Some commonly used metaphors are the following:

- White board
- Shopping cart
- Desktop
- Editor's work bench
- White page
- Yellow page
- Office cabinet
- Post box
- Bulletin board
- Visitor's Book

Task and ob ject modelling

A task is a human activity intended to achieve some goals. Examples of task goals can be as follows:

- Reserve an airline seat
- Buy an item
- Transfer money from one account to another
- Book a cargo for transmission to an address

A task model is an abstract model of the structure of a task. A task model should show the structure of the subtasks that the user needs to perform to achieve the overall task goal. Each task can be modeled as a hierarchy of subtasks. A task model can be drawn using a graphical notation similar to the activity network model we discussed in Chapter 3. Tasks can be drawn as boxes with lines showing how a task is broken down into subtasks. An underlined task box would mean that no further decomposition of the task is required. An example of decomposition of a task into subtasks is shown in

Figure 9.7.

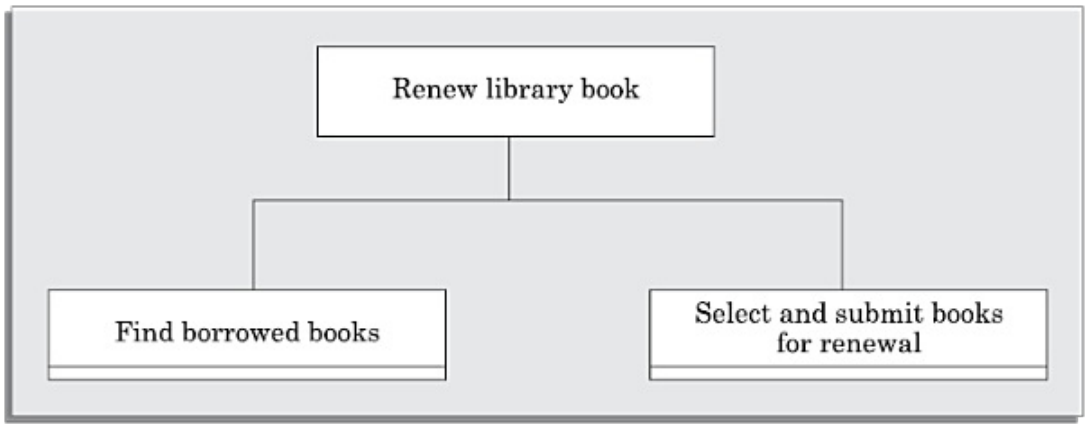

Figure 9.7: Decomposition of a task into subtasks.

Identification of the user objects forms the basis of an object-based design. A user object model is a model of business objects which the end-users believe that they are interacting with. The objects in a library software may be books, journals, members, etc. The objects in the supermarket automation software may be items, bills, indents, shopping list, etc. The state diagram for an object can be drawn using a notation similar to that used by UML (see Section 7.8). The state diagram of an object model can be used to determine which menu items should be dimmed in a state. An example state chart diagram for an order object is shown in Figure 9.8.

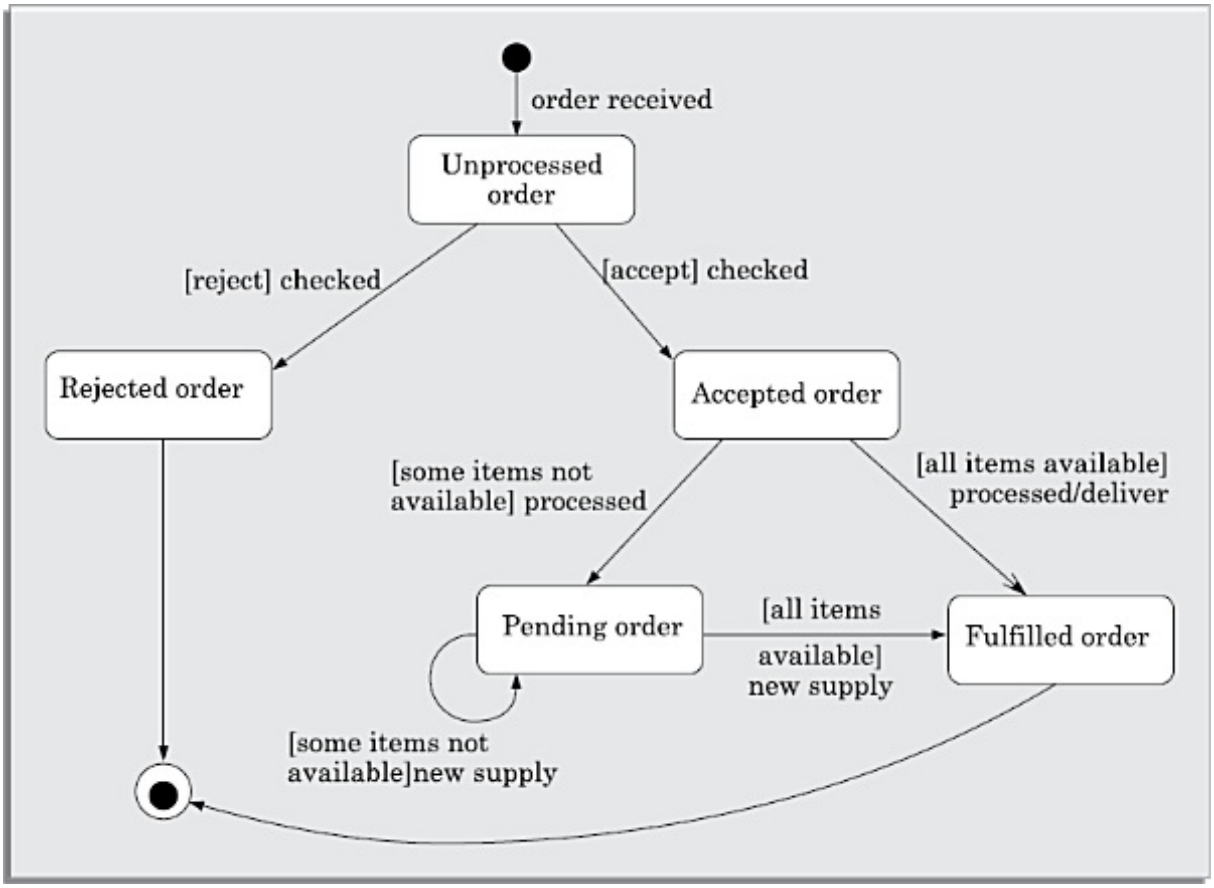

Figure 9.8: State chart diagram for an order object.

Metaphor selection

The first place one should look for while trying to identify the candidate metaphors is the set of parallels to objects, tasks, and terminologies of the use cases. If no obvious metaphors can be found, then the designer can fall back on the metaphors of the physical world of concrete objects. The appropriateness of each candidate metaphor should be tested by restating the objects and tasks of the user interface model in terms of the metaphor. Another criterion that can be used to judge metaphors is that the metaphor should be as simple as possible, the operations using the metaphor should be clear and coherent and it should fit with the users' 'common sense' knowledge. For example, it would indeed be very awkward and a nuisance for the users if the scissor metaphor is used to glue different items.

Example 9.1 We need to develop the interface for a web-based pay-order shop, where the users can examine the contents of the shop through a web browser and can order them.

Several metaphors are possible for different parts of this problem as follows:

- Different items can be picked up from racks and examined. The user can request for the **catalog** associated with the items by clicking on the item.
- Related items can be picked from the drawers of an item cabinet.
- The items can be organised in the form of a book, similar to the way information abo u t electronic components are organised in a semiconductor hand book.

Once the users make up their mind about an item they wish to buy, they can put them into a shopping cart.

Interaction design and rough layout

The interaction design involves mapping the subtasks into appropriate controls, and other widgets such as forms, text box, etc. This involves making a choice from a set of available components that would best suit the subtask. Rough layout concerns how the controls, an other widgets to be organised in windows.

Detailed presentation and graphics design

Each window should represent either an object or many objects that have a clear relationship to each other. At one extreme, each object view could be in its own window. But, this is likely to lead to too much window opening, closing, moving, and resizing. At the other extreme, all the views could be placed in one window side-by-side, resulting in a very large window. This would force the user to move the cursor around the window to look for different objects.

GUI construction

Some of the windows have to be defined as modal dialogs. When a window is a modal dialog, no other windows in the application is accessible until the current window is closed. When a modal dialog is closed, the user is returned to the window from which the modal dialog was invoked. Modal dialogs are commonly used when an explicit confirmation or authorisation step is required for an action (e.g., confirmation of delete). Though use of modal dialogs are essential in some situations, overuse of modal dialogs reduces user flexibility. In particular, sequences of modal dialogs should be avoided.

User interface inspection

Nielson [Niel94] studied common usability problems and built a check list of points which can be easily checked for an interface. The following check list is based on the work of Nielson [Niel94]:

Visibility of the system status: The system should as far as possible keep the user informed about the status of the system and what is going on. For example, it should not be the case that a user gives a command and keeps waiting, wondering whether the system has crashed and he should reboot the system or that the results shall appear after some more time.

Match between the system and the real world: The system should speak the user's language with words, phrases, and concepts familiar to that used by the user, rather than using system-oriented terms.

Undoing mistakes: The user should feel that he is in control rather than feeling helpless or to be at the control of the system. An important step toward this is that the users should be able to undo and redo operations.

Consistency: The users should not have to wonder whether different words, concepts, and operations mean the same thing in different situations.

Recognition rather than recall: The user should not have to recall information which was presented in another screen. All data and instructions should be visible on the screen for selection by the user.

Support for multiple skill levels: Provision of accelerators for experienced users allows them to efficiently carry out the actions they most frequently require to perform.

Aesthetic and minimalist design: Dialogs and screens should not contain information which are irrelevant and are rarely needed. Every extra unit of information in a dialog or screen competes with the relevant units and diminishes their visibility.

Help and error messages: These should be expressed in plain language (no codes), precisely indicating the problem, and constructively suggesting a solution.

Error prevention: Error possibilities should be minimised. A key principle in this regard is to prevent the user from entering wrong values. In situations where a choice has to be made from among a discrete set of values, the control should present only the valid values using a drop-down list, a set of option buttons or a similar multichoice control. When a specific format is required for attribute data, the entered data should be validated when the user attempts to submit the data.

SUMMARY

- In this chapter, we first discussed some important concepts associated with user interface design and suggested a few desirable properties that a good user interface should possess.
- The current user interface development-style is component-based. Component-based user interfaces enhance speed of learning and also reduce the interface development effort. Component-based interface development makes the users familiar with standard ways of interacting with an interface. The users can easily extend their knowledge of interacting with one interface to another, thereby reducing the learning time to a great extent.
- We then identified several primitive components which can be used to design graphical user interfaces.
- We discussed the basic concepts associated with window management systems and provided a brief overview of X-Window/Motif and Visual programming.
- We discussed the rudiments of a user interface design methodology which can be used to supplement the ingenuity of a designer.

EXERCISES

1. Choose the correct option:
   (a) Which of the following can be considered to provide the most accurate measure of the size of a user interface:
   (i) LOC of the GUI components
   (ii) Number of scenarios
   (iii) Number of windows
   (iv) Sizes of input and output data
   (b) Which of the following types of interfaces would the novice users find the easiest to use?
   (i) Direct manipulation interfaces
   (ii) Menu-based interfaces
   (iii) Command-based interfaces
   (iv) Combination of menu and command interfaces
   (c) Widgets in user interface terminology stand for:
   (i) Window objects
   (ii) Orphaned window
   (iii) What you see is what you get

(iv) Wily midget

2. List five desirable characteristics that a good user interface should possess.
3. Compare the relative advantages of textual and graphical user interfaces.
4. What is the difference between user guidance and on-line help system in the user interface of a software system? Discuss the different ways in which on-line help can be provided to a user while he is executing the software.
5. (a) Compare the relative advantages of command language, menu-based, and direct manipulation interfaces.
   (b) Suppose you have been asked to design the user interface of a large software product. Would you choose a menu-based, a direct manipulation, a command language-based, or a mixture of all these types of interfaces to develop the interface for your product? Justify your choice.
6. State **TRUE** or **FALSE** of the following. Support your answer with proper reasoning:
   (a) Visual programming style is restricted to user interface development only.
   (b) A modeless user interface is preferred for a software product that needs to support a large number of functionalities for the user.
   (c) Novice users normally prefer command language interfaces over both menu-based and iconic interfaces.
   (d) For modern user interfaces, LOC is an accurate measure of the size of the interface. (e) The look and feel of a window system is essentially determined by the window manager.
7. Discuss why several popular software packages support a command language in addition to menu-based and iconic user interfaces.
8. List the important advantages and disadvantages of a command language interface.
9. List the important advantages and disadvantages of a menu-based interface.
10. While developing the user interface for a software product, how can you accomodate users with different skill levels.
11. Compare the relative advantages of scrolling menu, hierarchical menu, and walking menu as techniques for organising user commands.

12. What do you understand by an iconic interface? Explain how you can issue commands using an iconic interface.
13. List the important advantages and disadvantages of a direct manipulation interface.
14. List the important GUI components using which you can develop a graphical user interface for any application.
15. What is the difference between a mode-based and a modeless user interface? What are their relative advantages? Which one would you use for developing the interface of a software product supporting a large number of commands? Justify your answer.
16. What is a window manager? Mention at least two important responsibilities of a window manager. Name some window managers commonly used in the Linux system.
17. What is a window management system (WMS)? Represent the main components of a WMS in a schematic diagram and briefly explain their roles.
18. What are the advantages of using a window management system for GUI design? Name some commercially available window management systems.
19. Briefly discuss the architecture of the X window system. What are the important advantages of using the X window system for developing graphical user interfaces?
20. Write five sentences to highlight the important features of Visual C++?
21. What do you understand by a visual language? How do languages such as Visual C++ and Visual Basic let you do component-based user interface development?
22. What do you understand by component-based user interface development? What are the advantages of component-based user interface development?
23. Why a count of the different screens of the GUI of an application may not be an accurate measure of the size of the user interface? Suggest a more accurate measure of the size of the user interface of an application. Explain how it overcomes the difficulties with the number of screens measure.
24. Distinguish between a "user-centered design" and "design by users." Examine the pros and cons of these two approaches to user interface

design.

25. Suppose the customer feedback for a product that you have developed is "too difficult to learn". Explain how the speed of learning of the user interface of the product can be increased.
26. Distinguish between procedural and object-oriented user interface. Which type of interface is more usable? Justify your answer.
27. What do you understand by "look and feel" of a window. Which component of an operating system determines the look and feel of the windows. Explain your answer.
28. Design and develop a graphical user interface for the graphical editor software described in Exercise 6.18.
29. Prepare a check list of at least five inspection items for effective inspection of any user interface.
30. Study the user interface of some popular software products such as Word, Powerpoint, Excel, OpenOffice, Gimp, etc. and identify the metaphors used. Also, examine how tasks are broken up into subtasks and closure achieved.
31. What do you understand by a metaphor in the context of user interface design? Why is it advantageous to design a user interface based on metaphors? List a few metaphors which can be used for user interface design.
32. What do you understand by a modal dialog? Why are these required? Why should the use of too many modal dialogs in an interface design be avoided?
33. How does the human cognition capabilities and limitations influence human-computer user interface designing?
34. Distinguish between a user-centric interface design and interface design by users.
35. Is there any difference between designing a software and model building based on the requirements of the software?
36. Define a metric using which the size of the user interface part of a software can be measured.

1 Dictionary meaning: figure of speech in which a word or phrase literally denoting one kind of object or idea is used in place of another to suggest a likeness or analogy between them as in drowning in money.

2 Small pictures.

Chapter 10

CODING AND TESTING

In this chapter, we will discuss the coding and testing phases of the software life cycle.

Coding is undertaken once the design phase is complete and the design documents have been successfully reviewed.

In the coding phase, every module specified in the design document is coded and unit tested. During unit testing, each module is tested in isolation from other modules. That is, a module is tested independently as and when its coding is complete.

After all the modules of a system have been coded and unit tested, the integration and system testing phase is undertaken.

Integration and testing of modules is carried out according to an integration plan. The integration plan, according to which different modules are integrated together, usually envisages integration of modules through a number of steps. During each integration step, a number of modules are added to the partially integrated system and the resultant system is tested. The full product takes shape only after all the modules have been integrated together. System testing is conducted on the full product. During system testing, the product is tested against its requirements as recorded in the SRS document.

We had already pointed out in Chapter 2 that testing is an important phase in software development and typically requires the maximum effort among all the development phases. Usually, testing of a professional software is carried out using a large number of test cases. It is usually the case that many of the different test cases can be executed in parallel by different team members. Therefore, to reduce the testing time, during the testing phase the largest manpower (compared to all other life cycle phases) is deployed. In a typical development organisation, at any time, the maximum number of software

engineers can be found to be engaged in testing activities. It is not very surprising then that in the software industry there is always a large demand for software test engineers. However, many novice engineers bear the wrong impression that testing is a secondary activity and that it is intellectually not as stimulating as the activities associated with the other development phases.

> Over the years, the general perception of testing as monkeys typing in random data and trying to crash the system has changed. Now testers are looked upon as masters of specialised concepts, techniques, and tools.

As we shall soon realize, testing a software product is as much challenging as initial development activities such as specifications, design, and coding. Moreover, testing involves a lot of creative thinking.

In this Chapter, we first discuss some important issues associated with the activities undertaken in the coding phase. Subsequently, we focus on various types of program testing techniques for procedural and object-oriented programs.

10.1 CODING

The input to the coding phase is the design document produced at the end of the design phase. Please recollect that the design document contains not only the high-level design of the system in the form of a module structure (e.g., a structure chart), but also the detailed design. The detailed design is usually documented in the form of module specifications where the data structures and algorithms for each module are specified. During the coding phase, different modules identified in the design document are coded according to their respective module specifications. We can describe the overall objective of the coding phase to be the following.

> The objective of the coding phase is to transform the design of a system into code in a high-level language, and then to unit test this code.

Normally, good software development organisations require their programmers to adhere to some well-defined and standard style of coding which is called their coding standard. These software development organisations formulate their own coding standards that suit them the most, and require their developers to follow the standards rigorously because of the significant business advantages it offers. The main advantages of adhering to a standard style of coding are the following:

- A coding standard gives a uniform appearance to the codes written by different engineers.
- It facilitates code understanding and code reuse.
- It promotes good programming practices.

A coding standard lists several rules to be followed during coding, such as the way variables are to be named, the way the code is to be laid out, the error return conventions, etc. Besides the coding standards, several coding guidelines are also prescribed by software companies. But, what is the difference between a coding guideline and a coding standard?

> It is mandatory for the programmers to follow the coding standards. Compliance of their code to coding standards is verified during code inspection. Any code that does not conform to the coding standards is rejected during code review and the code is reworked by the concerned programmer. In contrast, coding guidelines provide some general suggestions regarding the coding style to be followed but leave the actual implementation of these guidelines to the discretion of the individual developers.

After a module has been coded, usually code review is carried out to ensure that the coding standards are followed and also to detect as many errors as possible before testing. It is important to detect as many errors as possible during code reviews, because reviews are an efficient way of removing errors from code as compared to defect elimination using testing. We first discuss a few representative coding standards and guidelines. Subsequently, we discuss code review techniques. We then discuss software documentation in Section 10.3.

10.1.1 Coding Standards and Guidelines

Good software development organisations usually develop their own coding standards and guidelines depending on what suits their organisation best and based on the specific types of software they develop. To give an idea about the types of coding standards that are being used, we shall only list some general coding standards and guidelines that are commonly adopted by many software development organisations, rather than trying to provide an exhaustive list.

Representative coding standards

Rules for limiting the use of globals: These rules list what types of data can be declared global and what cannot, with a view to limit the data that needs to be defined with global scope.

Standard headers for different modules: The header of different modules should have standard format and information for ease of understanding and maintenance. The following is an example of header format that is being used in some companies:

- Name of the module.
- Date on which the module was created.
- Author's name.
- Modification history.
- Synopsis of the module. This is a small writeup about what the module does.
- Different functions supported in the module, along with their input/output parameters.
- Global variables accessed/modified by the module.

Naming conventions for global variables, local variables, and constant identifiers: A popular naming convention is that variables are named using mixed case lettering. Global variable names would always start with a capital letter (e.g., GlobalData) and local variable names start with small letters (e.g., localData). Constant names should be formed using capital letters only (e.g., CONSTDATA).

Conventions regarding error return values and exception handling mechanisms: The way error conditions are reported by different functions in a program should be standard within an organisation. For example, all functions while encountering an error condition should either return a 0 or 1 consistently, independent of which programmer has written the code. This facilitates reuse and debugging.

Representative coding guidelines: The following are some representative coding guidelines that are recommended by many software development organisations. Wherever necessary, the rationale behind these guidelines is also mentioned.

Do not use a coding style that is too clever or too difficult to understand: Code should be easy to understand. Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code. Clever coding can obscure meaning of the code and reduce code understandability; thereby making maintenance and debugging difficult and expensive.

Avoid obscure side effects: The side effects of a function call include modifications to the parameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not obvious from a casual examination of the code. Obscure side effects make it difficult to understand a piece of code. For example, suppose the value of a global variable is changed or some file I/O is performed obscurely in a called module. That is, this is difficult to infer from the function's name and header information. Then, it would be really hard to understand the code.

Do not use an identifier for multiple purposes: Programmers often use the same identifier to denote several temporary entities. For example, some programmers make use of a temporary loop variable for also computing and storing the final result. The rationale that they give for such multiple use of variables is memory efficiency, e.g., three variables use up three memory locations, whereas when the same variable is used for three different purposes, only one memory location is used. However, there are several things wrong with this approach and hence should be avoided. Some of the problems caused by the use of a variable for multiple purposes are as follows:

- Each variable should be given a descriptive name indicating its purpose. This is not possible if an identifier is used for multiple purposes. Use of a variable for multiple purposes can lead to confusion and make it difficult for somebody trying to read and understand the code.
- Use of variables for multiple purposes usually makes future enhancements more difficult. For example, while changing the final computed result from integer to float type, the programmer might subsequently notice that it has also been used as a temporary loop variable that cannot be a float type.

Code should be well-documented: As a rule of thumb, there should be at least one comment line on the average for every three source lines of code.

Length of any function should not exceed 10 source lines: A lengthy function is usually very difficult to understand as it probably has a large number of variables and carries out many different types of computations. For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.

Do not use GO TO statements: Use of GO TO statements makes a program

unstructured. This makes the program very difficult to understand, debug, and maintain.

10.2 CODE REVIEW

Testing is an effective defect removal mechanism. However, testing is applicable to only executable code. Review is a very effective technique to remove defects from source code. In fact, review has been acknowledged to be more cost-effective in removing defects as compared to testing. Over the years, review techniques have become extremely popular and have been generalised for use with other work products.

Code review for a module is undertaken after the module successfully compiles. That is, all the syntax errors have been eliminated from the module. Obviously, code review does not target to design syntax errors in a program, but is designed to detect logical, algorithmic, and programming errors. Code review has been recognised as an extremely cost-effective strategy for eliminating coding errors and for producing high quality code.

The reason behind why code review is a much more cost-effective strategy to eliminate errors from code compared to testing is that reviews directly detect errors. On the other hand, testing only helps detect failures and significant effort is needed to locate the error during debugging.

The rationale behind the above statement is explained as follows. Eliminating an error from code involves three main activities—testing, debugging, and then correcting the errors. Testing is carried out to detect if the system fails to work satisfactorily for certain types of inputs and under certain circumstances. Once a failure is detected, debugging is carried out to locate the error that is causing the failure and to remove it. Of the three testing activities, debugging is possibly the most laborious and time consuming activity. In code inspection, errors are directly detected, thereby saving the significant effort that would have been required to locate the error.

Normally, the following two types of reviews are carried out on the code of a module:

- Code inspection.
- Code walkthrough.

The procedures for conduction and the final objectives of these two review techniques are very different. In the following two subsections, we discuss

these two code review techniques.

10.2.1 Code Walkthrough

Code walkthrough is an informal code analysis technique. In this technique, a module is taken up for review after the module has been coded, successfully compiled, and all syntax errors have been eliminated. A few members of the development team are given the code a couple of days before the walkthrough meeting. Each member selects some test cases and simulates execution of the code by hand (i.e., traces the execution through different statements and functions of the code).

The main objective of code walkthrough is to discover the algorithmic and logical errors in the code.

The members note down their findings of their walkthrough and discuss those in a walkthrough meeting where the coder of the module is present.

Even though code walkthrough is an informal analysis technique, several guidelines have evolved over the years for making this naive but useful analysis technique more effective. These guidelines are based on personal experience, common sense, several other subjective factors. Therefore, these guidelines should be considered as examples rather than as accepted rules to be applied dogmatically. Some of these guidelines are following:

- The team performing code walkthrough should not be either too big or too small. Ideally, it should consist of between three to seven members.
- Discussions should focus on discovery of errors and avoid deliberations on how to fix the discovered errors.
- In order to foster co-operation and to avoid the feeling among the engineers that they are being watched and evaluated in the code walkthrough meetings, managers should not attend the walkthrough meetings.

10.2.2 Code Inspection

During code inspection, the code is examined for the presence of some common programming errors. This is in contrast to the hand simulation of code execution carried out during code walkthroughs. We can state the principal aim of the code inspection to be the following:

The principal aim of code inspection is to check for the presence of some common types of errors that usually creep into code due to programmer mistakes and oversights and to check whether coding standards have been adhered to.

The inspection process has several beneficial side effects, other than finding errors. The programmer usually receives feedback on programming style, choice of algorithm, and programming techniques. The other participants gain by being exposed to another programmer's errors.

As an example of the type of errors detected during code inspection, consider the classic error of writing a procedure that modifies a formal parameter and then calls it with a constant actual parameter. It is more likely that such an error can be discovered by specifically looking for this kinds of mistakes in the code, rather than by simply hand simulating execution of the code. In addition to the commonly made errors, adherence to coding standards is also checked during code inspection.

Good software development companies collect statistics regarding different types of errors that are commonly committed by their engineers and identify the types of errors most frequently committed. Such a list of commonly committed errors can be used as a checklist during code inspection to look out for possible errors.

Following is a list of some classical programming errors which can be checked during code inspection:

- Use of uninitialised variables.
- Jumps into loops.
- Non-terminating loops.
- Incompatible assignments.
- Array indices out of bounds.
- Improper storage allocation and deallocation.
- Mismatch between actual and formal parameter in procedure calls.
- Use of incorrect logical operators or incorrect precedence among operators.
- Improper modification of loop variables.
- Comparison of equality of floating point values.
- Dangling reference caused when the referenced memory has not been allocated.

10.2.3 Clean Room Testing

Clean room testing was pioneered at IBM. This type of testing relies

heavily on walkthroughs, inspection, and formal verification. The programmers are not allowed to test any of their code by executing the code other than doing some syntax testing using a compiler. It is interesting to note that the term cleanroom was first coined at IBM by drawing analogy to the semiconductor fabrication units where defects are avoided by manufacturing in an ultra-clean atmosphere.

This technique reportedly produces documentation and code that is more reliable and maintainable than other development methods relying heavily on code execution-based testing. The main problem with this approach is that testing effort is increased as walkthroughs, inspection, and verification are time consuming for detecting all simple errors. Also testing- based error detection is efficient for detecting certain errors that escape manual inspection.

10.3 SOFTWARE DOCUMENTATION

When a software is developed, in addition to the executable files and the source code, several kinds of documents such as users' manual, software requirements specification (SRS) document, design document, test document, installation manual, etc., are developed as part of the software engineering process. All these documents are considered a vital part of any good software development practice. Good documents are helpful in the following ways:

- Good documents help enhance understandability of code. As a result, the availability of good documents help to reduce the effort and time required for maintenance.
- Documents help the users to understand and effectively use the system.
- Good documents help to effectively tackle the manpower turnover[1] problem. Even when an engineer leaves the organisation, and a new engineer comes in, he can build up the required knowledge easily by referring to the documents.
- Production of good documents helps the manager to effectively track the progress of the project. The project manager would know that some measurable progress has been achieved, if the results of some pieces of work has been documented and the same has been reviewed.

Different types of software documents can broadly be classified into the following:

Internal documentation: These are provided in the source code itself.

External documentation: These are the supporting documents such as SRS document, installation document, user manual, design document, and test document.

We discuss these two types of documentation in the next section.

10.3.1 Internal Documentation

Internal documentation is the code comprehension features provided in the source code itself. Internal documentation can be provided in the code in several forms. The important types of internal documentation are the following:

- Comments embedded in the source code.
- Use of meaningful variable names.
- Module and function headers.
- Code indentation.
- Code structuring (i.e., code decomposed into modules and functions).
- Use of enumerated types.
- Use of constant identifiers.
- Use of user-defined data types.

Out of these different types of internal documentation, which one is the most valuable for understanding a piece of code?

Careful experiments suggest that out of all types of internal documentation, meaningful variable names is most useful while trying to understand a piece of code.

The above assertion, of course, is in contrast to the common expectation that code commenting would be the most useful. The research finding is obviously true when comments are written without much thought. For example, the following style of code commenting is not much of a help in understanding the code.

```
a=10; /* a made 10 */
```

A good style of code commenting is to write to clarify certain non-obvious aspects of the working of the code, rather than cluttering the code with trivial comments. Good software development organisations usually ensure good internal documentation by appropriately formulating their coding standards

and coding guidelines. Even when a piece of code is carefully commented, meaningful variable names has been found to be the most helpful in understanding the code.

10.3.2 External Documentation

External documentation is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test document, etc. A systematic software development style ensures that all these documents are of good quality and are produced in an orderly fashion.

An important feature that is requierd of any good external documentation is consistency with the code. If the different documents are not consistent, a lot of confusion is created for somebody trying to understand the software. In other words, all the documents developed for a product should be up-to-date and every change made to the code should be reflected in the relevant external documents. Even if only a few documents are not up-to-date, they create inconsistency and lead to confusion. Another important feature required for external documents is proper understandability by the category of users for whom the document is designed. For achieving this, Gunning's fog index is very useful. We discuss this next.

Gunning's fog index

Gunning's fog index (developed by Robert Gunning in 1952) is a metric that has been designed to measure the readability of a document. The computed metric value (fog index) of a document indicates the number of years of formal education that a person should have, in order to be able to comfortably understand that document. That is, if a certain document has a fog index of 12, any one who has completed his 12th class would not have much difficulty in understanding that document.

The Gunning's fog index of a document D can be computed as follows:

$$\text{fog}(D) = 0.4 \times \left(\frac{\text{words}}{\text{sentences}}\right) + \text{per cent of words having 3 or more syllables}$$

Observe that the fog index is computed as the sum of two different factors. The first factor computes the average number of words per sentence (total number of words in the document divided by the total number of sentences). This factor therefore accounts for the common observation that long sentences are difficult to understand. The second factor measures the percentage of complex words in the document. Note that a syllable is a group

of words that can be independently pronounced. For example, the word "sentence" has three syllables ("sen", "ten", and "ce"). Words having more than three syllables are complex words and presence of many such words hamper readability of a document.

Example 10.1 Consider the following sentence: "The Gunning's fog index is based on the premise that use of short sentences and simple words makes a document easy to understand." Calculate its Fog index.

The fog index of the above example sentence is

$$0.4 \square (23/1) + (4/23) \square 100 = 26$$

If a users' manual is to be designed for use by factory workers whose educational qualification is class 8, then the document should be written such that the Gunning's fog index of the document does not exceed 8.

10.4 TESTING

The aim of program testing is to help realiseidentify all defects in a program. However, in practice, even after satisfactory completion of the testing phase, it is not possible to guarantee that a program is error free. This is because the input data domain of most programs is very large, and it is not practical to test the program exhaustively with respect to each value that the input can assume. Consider a function taking a floating point number as argument. If a tester takes 1sec to type in a value, then even a million testers would not be able to exhaustively test it after trying for a million number of years. Even with this obvious limitation of the testing process, we should not underestimate the importance of testing. We must remember that careful testing can expose a large percentage of the defects existing in a program, and therefore provides a practical way of reducing defects in a system.

10.4.1 Basic Concepts and Terminologies

In this section, we will discuss a few basic concepts in program testing on which our subsequent discussions on program testing would be based.

How to test a program?

Testing a program involves executing the program with a set of test inputs and observing if the program behaves as expected. If the

program fails to behave as expected, then the input data and the conditions under which it fails are noted for later debugging and error correction. A highly simplified view of program testing is schematically shown in Figure 10.1. The tester has been shown as a stick icon, who inputs several test data to the system and observes the outputs produced by it to check if the system fails on some specific inputs. Unless the conditions under which a software fails are noted down, it becomes difficult for the developers to reproduce a failure observed by the testers. For examples, a software might fail for a test case only when a network connection is enabled.

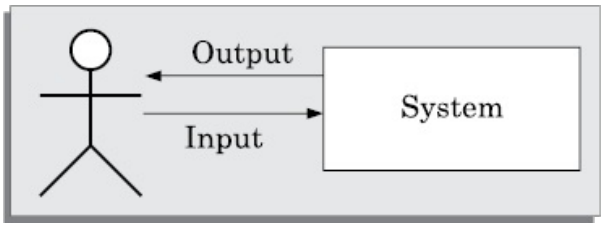

Figure 10.1: A simplified view of program testing.

Terminologies

As is true for any specialised domain, the area of software testing has come to be associated with its own set of terminologies. In the following, we discuss a few important terminologies that have been standardised by the IEEE Standard Glossary of Software Engineering Terminology [IEEE90]:

- A **mistake** is essentially any programmer action that later shows up as an incorrect result during program execution. A programmer may commit a mistake in almost any development activity. For example, during coding a programmer might commit the mistake of not initializing a certain variable, or might overlook the errors that might arise in some exceptional situations such as division by zero in an arithmetic operation. Both these mistakes can lead to an incorrect result.
- An **error** is the result of a mistake committed by a developer in any of the development activities. Among the extremely large variety of errors that can exist in a program. One example of an error is a call made to a wrong function.

The terms error, fault, bug, and defect are considered to be synonyms in the area of

program testing.

Though the terms error, fault, bug, and defect are all used interchangeably by the program testing community. Please note that in the domain of hardware testing, the term fault is used with a slightly different connotation [IEEE90] as compared to the terms error and bug.

Example 10.2 Can a designer's mistake give rise to a program error? Give an example of a designer's mistake and the corresponding program error.

Answer: Yes, a designer's mistake give rise to a program error. For example, a requirement might be overlooked by the designer, which can lead to it being overlooked in the code as well.

- A **failure** of a program essentially denotes an incorrect behaviour exhibited by the program during its execution. An incorrect behaviour is observed either as an incorrect result produced or as an inappropriate activity carried out by the program. Every failure is caused by some bugs present in the program. In other words, we can say that every software failure can be traced to some bug or other present in the code. The number of possible ways in which a program can fail is extremely large. Out of the large number of ways in which a program can fail, in the following we give three randomly selected examples:

– The result computed by a program is 0, when the correct result is 10.
– A program crashes on an input.
– A robot fails to avoid an obstacle and collides with it.

It may be noted that mere presence of an error in a program code may not necessarily lead to a failure during its execution.

Example 10.3 Give an example of a program error that may not cause any failure.

Answer: Consider the following C program segment:

```
int markList[1:10]; /* mark list of 10 students*/
int roll;           /* student roll number*/
       ...
if(roll>0)
       markList[roll]=mark;
else
       markList[roll]=0;
```

In the above code, if the variable roll assumes zero or some negative value

under some circumstances, then an array index out of bound type of error would result. However, it may be the case that for all allowed input values the variable roll is always assigned positive values. Then, the else clause is unreachable and no failure would occur. Thus, even if an error is present in the code, it does not show up as an error since it is unreachable for normal input values.

Explanation: An array index out of bound type of error is said to occur, when the array index variable assumes a value beyond the array bounds.

- A **test case** is a triplet [I , S, R], where I is the data input to the program under test, S is the state of the program at which the data is to be input, and R is the result expected to be produced by the program. The state of a program is also called its execution mode. As an example, consider the different execution modes of a certain text editor software. The text editor can at any time during its execution assume any of the following execution modes—edit, view, create, and display. In simple words, we can say that a test case is a set of test inputs, the mode in which the input is to be applied, and the results that are expected during and after the execution of the test case.

An example of a test case is—[input: "abc", state: edit, result: abc is displayed], which essentially means that the input abc needs to be applied in the edit mode, and the expected result is that the string abc would be displayed.

- A **test scenario** is an abstract test case in the sense that it only identifies the aspects of the program that are to be tested without identifying the input, state, or output. A test case can be said to be an implementation of a test scenario. In the test case, the input, output, and the state at which the input would be applied is designed such that the scenario can be executed. An important automatic test case design strategy is to first design test scenarios through an analysis of some program abstraction (model) and then implement the test scenarios as test cases.
- A **test script** is an encoding of a test case as a short program. Test scripts are developed for automated execution of the test cases.
- A test case is said to be a **positive test case** if it is designed to test whether the software correctly performs a required functionality. A test

case is said to be **negative test case**, if it is designed to test whether the software carries out something, that is not required of the system. As one example each of a positive test case and a negative test case, consider a program to manage user login. A positive test case can be designed to check if a login system validates a user with the correct user name and password. A negative test case in this case can be a test case that checks whether the the login functionality validates and admits a user with wrong or bogus login user name or password.

- A **test suite** is the set of all test that have been designed by a tester to test a given program.
- **Testability** of a requirement denotes the extent to which it is possible to determine whether an implementation of the requirement conforms to it in both functionality and performance. In other words, the testability of a requirement is the degree to which an implementation of it can be adequately tested to determine its conformance to the requirement.

Example 10.4 Suppose two programs have been written to implement essentially the same functionality. How can you determine which of these is more testable?

Answer: A program is more testable, if it can be adequately tested with less number of test cases. Obviously, a less complex program is more testable. The complexity of a program can be measured using several types of metrics such as number of decision statements used in the program. Thus, a more testable program should have a lower structural complexity metric.

- A **failure mode** of a software denotes an observable way in which it can fail. In other words, all failures that have similar observable symptoms, constitute a failure mode. As an example of the failure modes of a software, consider a railway ticket booking software that has three failure modes—failing to book an available seat, incorrect seat booking (e.g., booking an already booked seat), and system crash.
- **Equivalent faults** denote two or more bugs that result in the system failing in the same failure mode. As an example of equivalent faults, consider the following two faults in C language—division by zero and illegal memory access errors. These two are equivalent faults, since each of these leads to a program crash.

Verification versus validation

The objectives of both verification and validation techniques are very similar since both these techniques are designed to help remove errors in a software. In spite of the apparent similarity between their objectives, the underlying principles of these two bug detection techniques and their applicability are very different. We summarise the main differences between these two techniques in the following:

- Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase; whereas validation is the process of determining whether a fully developed software conforms to its requirements specification. Thus, the objective of verification is to check if the work products produced after a phase conform to that which was input to the phase. For example, a verification step can be to check if the design documents produced after the design step conform to the requirements specification. On the other hand, validation is applied to the fully developed and integrated software to check if it satisfies the customer's requirements.
- The primary techniques used for verification include review, simulation, formal verification, and testing. Review, simulation, and testing are usually considered as informal verification techniques. Formal verification usually involves use of theorem proving techniques or use of automated tools such as a model checker. On the other hand, validation techniques are primarily based on product testing. Note that we have categorised testing both under program verification and validation. The reason being that unit and integration testing can be considered as verification steps where it is verified whether the code is as per the module and module interface specifications. On the other hand, system testing can be considered as a validation step where it is determined whether the fully developed code is as per its requirements specification.
- Verification does not require execution of the software, whereas validation requires execution of the software.
- Verification is carried out during the development process to check if the development activities are proceeding alright, whereas validation is carried out to check if the right as required by the customer has been developed.

We can therefore say that the primary objective of the verification steps are to determine whether the steps in product development are being carried out alright, whereas validation is carried out towards the end of the development process to determine whether the right product has been developed.

- Verification techniques can be viewed as an attempt to achieve phase containment of errors. Phase containment of errors has been acknowledged to be a cost-effective way to eliminate program bugs, and is an important software engineering principle. The principle of detecting errors as close to their points of commitment as possible is known as phase containment of errors. Phase containment of errors can reduce the effort required for correcting bugs. For example, if a design problem is detected in the design phase itself, then the problem can be taken care of much more easily than if the error is identified, say, at the end of the testing phase. In the later case, it would be necessary not only to rework the design, but also to appropriately redo the relevant coding as well as the system testing activities, thereby incurring higher cost.

While verification is concerned with phase containment of errors, the aim of validation is to check whether the deliverable software is error free.

We can consider the verification and validation techniques to be different types of bug filters. To achieve high product reliability in a cost-effective manner, a development team needs to perform both verification and validation activities. The activities involved in these two types of bug detection techniques together are called the "V and V" activities.

Based on the above discussions, we can conclude that:

Error detection techniques = Verification techniques + Validation techniques

Example 10.5 Is it at all possible to develop a highly reliable software, using validation techniques alone? If so, can we say that all verification techniques are redundant?

Answer: It is possible to develop a highly reliable software using validation techniques alone. However, this would cause the development cost to increase drastically. Verification techniques help achieve phase containment of errors and provide a means to cost-effectively remove bugs.

10.4.2 Testing Activities

Testing involves performing the following main activities:

Test suite design: The set of test cases using which a program is to be tested is designed possibly using several test case design techniques. We discuss a few important test case design techniques later in this Chapter.

Running test cases and checking the results to detect failures: Each test case is run and the results are compared with the expected results. A mismatch between the actual result and expected results indicates a failure. The test cases for which the system fails are noted down for later debugging.

Locate error: In this activity, the failure symptoms are analysed to locate the errors. For each failure observed during the previous activity, the statements that are in error are identified.

Error correction: After the error is located during debugging, the code is appropriately changed to correct the error.

The testing activities have been shown schematically in Figure 10.2. As can be seen, the test cases are first designed, the test cases are run to detect failures. The bugs causing the failure are identified through debugging, and the identified error is corrected.Of all the above mentioned testing activities, debugging often turns out to be the most time-consuming activity.

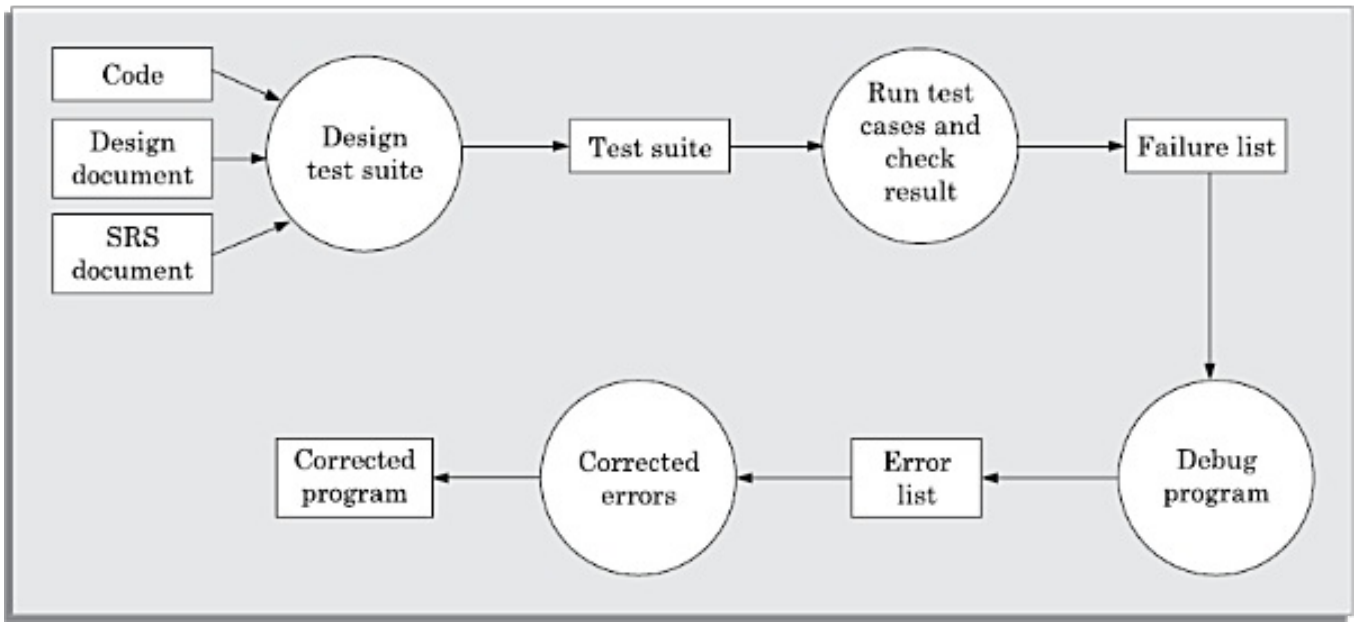

Figure 10.2: Testing process.

10.4.3 Why Design Test Cases?

Before discussing the various test case design techniques, we need to convince ourselves on the following question. Would it not be sufficient to test a software using a large number of random input values? Why design

test cases? The answer to this question—this would be very costly and at the same time very ineffective way of testing due to the following reasons:

> When test cases are designed based on random input data, many of the test cases do not contribute to the significance of the test suite, That is, they do not help detect any additional defects not already being detected by other test cases in the suite.

Testing a software using a large collection of randomly selected test cases does not guarantee that all (or even most) of the errors in the system will be uncovered. Let us try to understand why the number of random test cases in a test suite, in general, does not indicate of the effectiveness of testing. Consider the following example code segment which determines the greater of two integer values x and y. This code segment has a simple programming error:

```
if (x>y) max = x;
else max = x;
```

For the given code segment, the test suite {(x=3,y=2);(x=2,y=3)} can detect the error, whereas a larger test suite {(x=3,y=2);(x=4,y=3); (x=5,y=1)} does not detect the error. All the test cases in the larger test suite help detect the same error, while the other error in the code remains undetected. So, it would be incorrect to say that a larger test suite would always detect more errors than a smaller one, unless of course the larger test suite has also been carefully designed. This implies that for effective testing, the test suite should be carefully designed rather than picked randomly.

We have already pointed out that exhaustive testing of almost any non-trivial system is impractical due to the fact that the domain of input data values to most practical software systems is either extremely large or countably infinite. Therefore, to satisfactorily test a software with minimum cost, we must design a minimal test suite that is of reasonable size and can uncover as many existing errors in the system as possible. To reduce testing cost and at the same time to make testing more effective, systematic approaches have been developed to design a small test suite that can detect most, if not all failures.

> A minimal test suite is a carefully designed set of test cases such that each test case helps detect different errors. This is in contrast to testing using some random input values.

There are essentially two main approaches to systematically design test cases:

- Black-box approach
- White-box (or glass-box) approach

In the black-box approach, test cases are designed using only the functional specification of the software. That is, test cases are designed solely based on an analysis of the input/out behaviour (that is, functional behaviour) and does not require any knowledge of the internal structure of a program. For this reason, black-box testing is also known as functional testing. On the other hand, designing white-box test cases requires a thorough knowledge of the internal structure of a program, and therefore white-box testing is also called structural testing. Black- box test cases are designed solely based on the input-output behaviour of a program. In contrast, white-box test cases are based on an analysis of the code. These two approaches to test case design are complementary. That is, a program has to be tested using the test cases designed by both the approaches, and one testing using one approach does not substitute testing using the other.

10.4.4 Testing in the Large versus Testing in the Small

A software product is normally tested in three levels or stages:

- Unit testing
- Integration testing
- System testing

During unit testing, the individual functions (or units) of a program are tested.

Unit testing is referred to as testing in the small, whereas integration and system testing are referred to as testing in the large.

After testing all the units individually, the units are slowly integrated and tested after each step of integration (integration testing). Finally, the fully integrated system is tested (system testing). Integration and system testing are known as testing in the large.

Often beginners ask the question—"Why test each module (unit) in isolation first, then integrate these modules and test, and again test the integrated set of modules—why not just test the integrated set of modules once thoroughly?" The answer to this question is the following—There are two main reasons to it. First while testing a module, other modules with which this module needs to interface may not be ready. Moreover, it is

always a good idea to first test the module in isolation before integration because it makes debugging easier. If a failure is detected when an integrated set of modules is being tested, it would be difficult to determine which module exactly has the error.

In the following sections, we discuss the different levels of testing. It should be borne in mind in all our subsequent discussions that unit testing is carried out in the coding phase itself as soon as coding of a module is complete. On the other hand, integration and system testing are carried out during the testing phase.

10.5 UNIT TESTING

Unit testing is undertaken after a module has been coded and reviewed. This activity is typically undertaken by the coder of the module himself in the coding phase. Before carrying out unit testing, the unit test cases have to be designed and the test environment for the unit under test has to be developed. In this section, we first discuss the environment needed to perform unit testing.

Driver and stub modules

In order to test a single module, we need a complete environment to provide all relevant code that is necessary for execution of the module. That is, besides the module under test, the following are needed to test the module:

- The procedures belonging to other modules that the module under test calls.
- Non-local data structures that the module accesses.
- A procedure to call the functions of the module under test with appropriate parameters.

Modules required to provide the necessary environment (which either call or are called by the module under test) are usually not available until they too have been unit tested. In this context, stubs and drivers are designed to provide the complete environment for a module so that testing can be carried out.

Stub: The role of stub and driver modules is pictorially shown in Figure 10.3. A stub procedure is a dummy procedure that has the same I/O parameters as the function called by the unit under test but has a highly simplified

behaviour. For example, a stub procedure may produce the expected behaviour using a simple table look up mechanism.

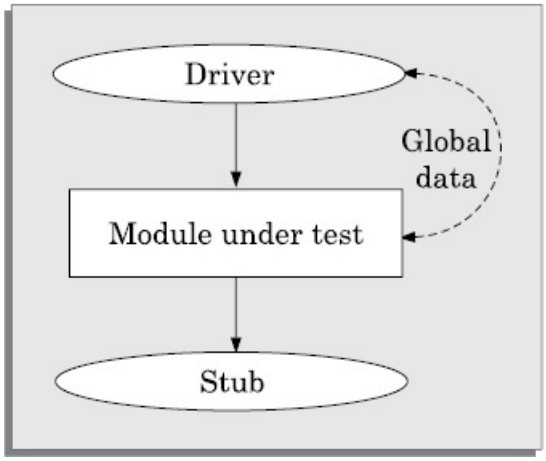

Figure 10.3: Unit testing with the help of driver and stub modules.

Driver: A driver module should contain the non-local data structures accessed by the module under test. Additionally, it should also have the code to call the different functions of the unit under test with appropriate parameter values for testing.

10.6 BLACK-BOX TESTING

In black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required. The following are the two main approaches available to design black box test cases:

- Equivalence class partitioning
- Boundary value analysis

In the following subsections, we will elaborate these two test case design techniques.

10.6.1 Equivalence Class Partitioning

In the equivalence class partitioning approach, the domain of input values to the program under test is partitioned into a set of equivalence classes. The partitioning is done such that for every input data belonging to the same equivalence class, the program behaves similarly.

The main idea behind defining equivalence classes of input data is that testing the code with any one value belonging to an equivalence class is as good as testing the

code with any other value belonging to the same equivalence class.

Equivalence classes for a unit under test can be designed by examining the input data and output data. The following are two general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes need to be defined. For example, if the equivalence class is the set of integers in the range 1 to 10 (i.e., [1,10]), then the invalid equivalence classes are $[-\infty,0]$, $[11,+\infty]$.
2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for the valid input values and another equivalence class for the invalid input values should be defined. For example, if the valid equivalence classes are {A,B,C}, then the invalid equivalence class is □-{A,B,C}, where □ is the universe of possible input values.

In the following, we illustrate equivalence class partitioning-based test case generation through four examples.

Example 10.6 For a software that computes the square root of an input integer that can assume values in the range of 0 and 5000. Determine the equivalence classes and the black box test suite.

Answer: There are three equivalence classes—The set of negative integers, the set of integers in the range of 0 and 5000, and the set of integers larger than 5000. Therefore, the test cases must include representatives for each of the three equivalence classes. A possible test suite can be: {–5,500,6000}.

Example 10.7 Design the equivalence class test cases for a program that reads two integer pairs (m_1, c_1) and (m_2, c_2) defining two straight lines of the form `y=mx+c`. The program computes the intersection point of the two straight lines and displays the point of intersection.

Answer: The equivalence classes are the following:

- Parallel lines ($m_1 = m_2$, $c_1 \neq c_2$)
- Intersecting lines ($m_1 \neq m_2$)
- Coincident lines ($m_1 = m_2$, $c_1 = c_2$)

Now, selecting one representative value from each equivalence class, we get the required equivalence class test suite {(2,2)(2,5),(5,5)(7,7), (10,10)

(10,10)}.

Example 10.8 Design equivalence class partitioning test suite for a function that reads a character string of size less than five characters and displays whether it is a palindrome.

Answer: The equivalence classes are the leaf level classes shown in Figure 10.4. The equivalence classes are palindromes, non-palindromes, and invalid inputs. Now, selecting one representative value from each equivalence class, we have the required test suite: {abc,aba,abcdef}.

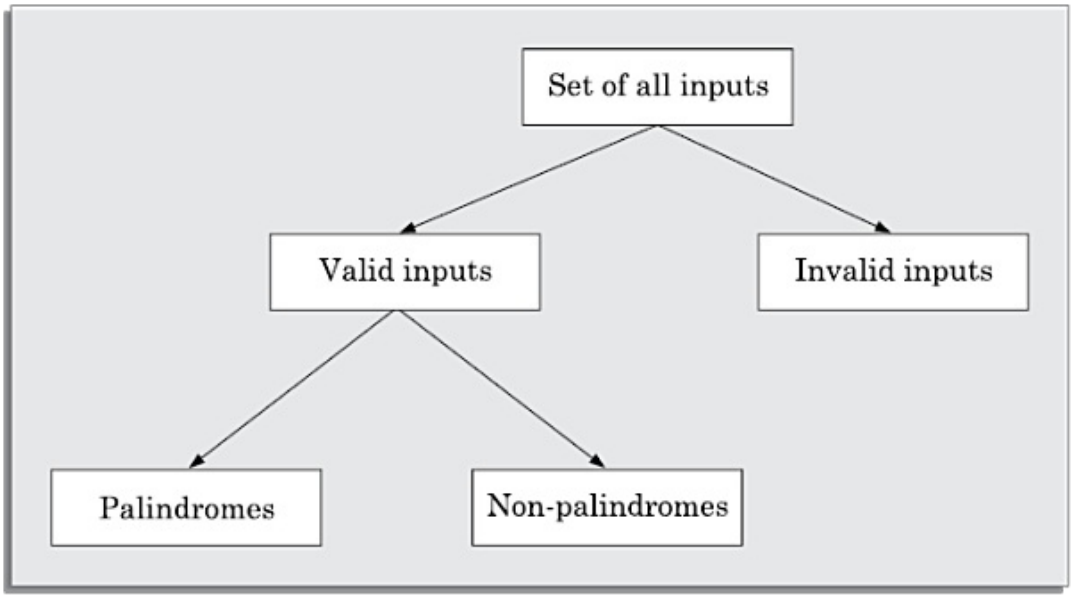

Figure 10.4: Equivalence classes for Example 10.6.

10.6.2 Boundary Value Analysis

A type of programming error that is frequently committed by programmers is missing out on the special consideration that should be given to the values at the boundaries of different equivalence classes of inputs. The reason behind programmers committing such errors might purely be due to psychological factors. Programmers often fail to properly address the special processing required by the input values that lie at the boundary of the different equivalence classes. For example, programmers may improperly use < instead of <=, or conversely <= for <, etc.

Boundary value analysis-based test suite design involves designing test cases using the values at the boundaries of different equivalence classes.

To design boundary value test cases, it is required to examine the equivalence classes to check if any of the equivalence classes contains a range of values. For those equivalence classes that are not a range of values

(i.e., consist of a discrete collection of values) no boundary value test cases can be defined. For an equivalence class that is a range of values, the boundary values need to be included in the test suite. For example, if an equivalence class contains the integers in the range 1 to 10, then the boundary value test suite is {0,1,10,11}.

Example 10.9 For a function that computes the square root of the integer values in the range of 0 and 5000, determine the boundary value test suite.

Answer: There are three equivalence classes—The set of negative integers, the set of integers in the range of 0 and 5000, and the set of integers larger than 5000. The boundary value-based test suite is: {0,-1,5000,5001}.

Example 10.10 Design boundary value test suite for the function described in Example 10.6.

Answer: The equivalence classes have been showed in Figure 10.5. There is a boundary between the valid and invalid equivalence classes. Thus, the boundary value test suite is {abcdefg, abcdef}.

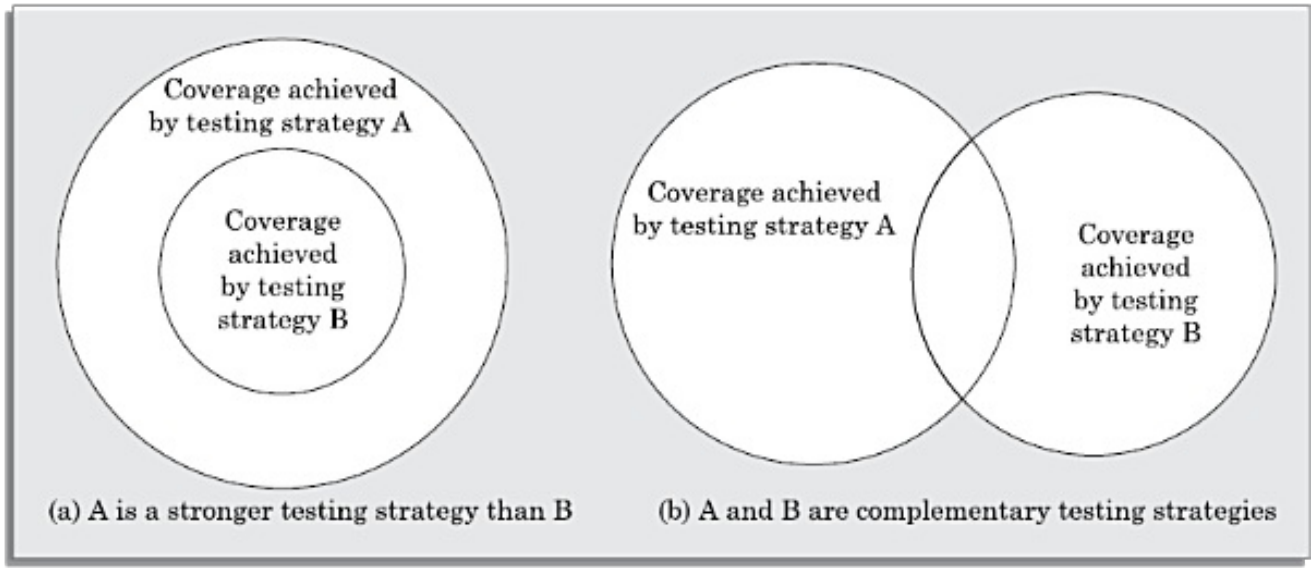

Figure 10.5: CFG for (a) sequence, (b) selection, and (c) iteration type of constructs.

10.6.3 Summary of the Black-box Test Suite Design Approach

We now summarise the important steps in the black-box test suite design approach:

- Examine the input and output values of the program.
- Identify the equivalence classes.
- Design equivalence class test cases by picking one representative

value from each equivalence class.

- Design the boundary value test cases as follows. Examine if any equivalence class is a range of values. Include the values at the boundaries of such equivalence classes in the test suite.

The strategy for black-box testing is intuitive and simple. For black-box testing, the most important step is the identification of the equivalence classes. Often, the identification of the equivalence classes is not straightforward. However, with little practice one would be able to identify all equivalence classes in the input data domain. Without practice, one may overlook many equivalence classes in the input data set. Once the equivalence classes are identified, the equivalence class and boundary value test cases can be selected almost mechanically.

10.7 WHITE-BOX TESTING

White-box testing is an important type of unit testing. A large number of white-box testing strategies exist. Each testing strategy essentially designs test cases based on analysis of some aspect of source code and is based on some heuristic. We first discuss some basic concepts associated with white-box testing, and follow it up with a discussion on specific testing strategies.

10.7.1 Basic Concepts

A white-box testing strategy can either be coverage-based or fault-based.

Fault-based testing

A fault-based testing strategy targets to detect certain types of faults. These faults that a test strategy focuses on constitutes the **fault model** of the strategy. An example of a fault-based strategy is mutation testing, which is discussed later in this section.

Coverage-based testing

A coverage-based testing strategy attempts to execute (or cover) certain elements of a program. Popular examples of coverage-based testing strategies are statement coverage, branch coverage, multiple condition coverage, and path coverage-based testing.

Testing criterion for coverage-based testing

A coverage-based testing strategy typically targets to execute (i.e., cover) certain program elements for discovering failures.

> The set of specific program elements that a testing strategy targets to execute is called the testing criterion of the strategy.

For example, if a testing strategy requires all the statements of a program to be executed at least once, then we say that the testing criterion of the strategy is statement coverage. We say that a test suite is adequate with respect to a criterion, if it covers all elements of the domain defined by that criterion.

Stronger versus weaker testing

We have mentioned that a large number of white-box testing strategies have been proposed. It therefore becomes necessary to compare the effectiveness of different testing strategies in detecting faults. We can compare two testing strategies by determining whether one is stronger, weaker, or complementary to the other.

> A white-box testing strategy is said to be stronger than another strategy, if the stronger testing strategy covers all program elements covered by the weaker testing strategy, and the stronger strategy additionally covers at least one program element that is not covered by the weaker strategy.

When none of two testing strategies fully covers the program elements exercised by the other, then the two are called complementary testing strategies. The concepts of stronger, weaker, and complementary testing are schematically illustrated in Figure 10.6. Observe in Figure 10.6(a) that testing strategy A is stronger than B since B covers only a proper subset of elements covered by B. On the other hand, Figure 10.6(b) shows A and B are complementary testing strategies since some elements of A are not covered by B and vice versa.

> If a stronger testing has been performed, then a weaker testing need not be carried out.

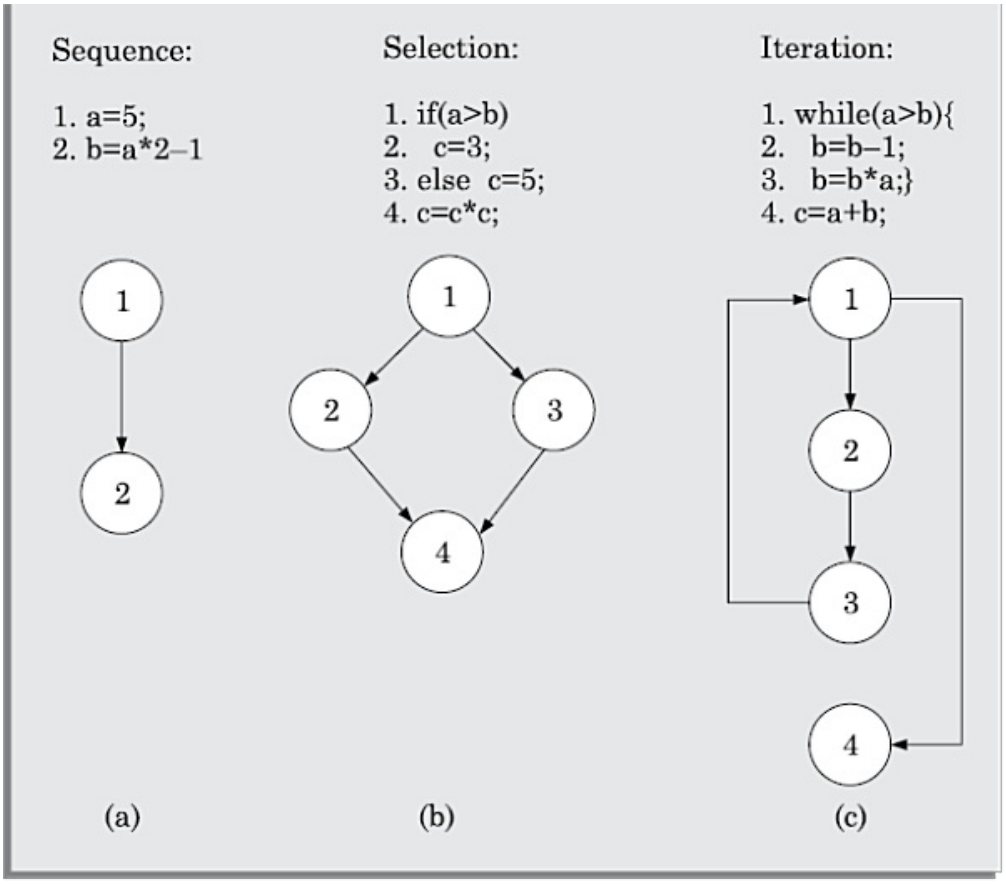

Figure 10.6: Illustration of stronger, weaker, and complementary testing strategies.

A test suite should, however, be enriched by using various complementary testing strategies.

We need to point out that coverage-based testing is frequently used to check the quality of testing achieved by a test suite. It is hard to manually design a test suite to achieve a specific coverage for a non-trivial program.

10.7.2 Statement Coverage

The statement coverage strategy aims to design test cases so as to execute every statement in a program at least once.

The principal idea governing the statement coverage strategy is that unless a statement is executed, there is no way to determine whether an error exists in that statement.

It is obvious that without executing a statement, it is difficult to determine whether it causes a failure due to illegal memory access, wrong result computation due to improper arithmetic operation, etc. It can however be pointed out that a weakness of the statement- coverage strategy is that executing a statement once and observing that it behaves properly for one

input value is no guarantee that it will behave correctly for all input values. Never the less, statement coverage is a very intuitive and appealing testing technique. In the following, we illustrate a test suite that achieves statement coverage.

Example 10.11 Design statement coverage-based test suite for the following Euclid's GCD computation program:

```
int computeGCD(x,y)
   int x,y;
{
    1 while (x != y){
    2 if (x>y) then
    3 x=x-y;
    4 else y=y-x;
    5 }
    6 return x;
}
```

Answer: To design the test cases for the statement coverage, the conditional expression of the `while` statement needs to be made true and the conditional expression of the `if` statement needs to be made both true and false. By choosing the test set {(x = 3, y = 3), (x = 4, y = 3), (x = 3, y = 4)}, all statements of the program would be executed at least once.

10.7.3 Branch Coverage

A test suite satisfies branch coverage, if it makes each branch condition in the program to assume true and false values in turn. In other words, for branch coverage each branch in the CFG representation of the program must be taken at least once, when the test suite is executed. Branch testing is also known as edge testing, since in this testing scheme, each edge of a program's control flow graph is traversed at least once.

Example 10.12 For the program of Example 10.11, determine a test suite to achieve branch coverage.

Answer: The test suite {(x = 3, y = 3), (x = 3, y = 2), (x = 4, y = 3), (x = 3, y = 4)} achieves branch coverage.

It is easy to show that branch coverage-based testing is a stronger testing than statement coverage-based testing. We can prove this by showing that branch coverage ensures statement coverage, but not vice versa.

Theorem 10.1 Branch coverage-based testing is stronger than statement coverage-based testing.

Proof: We need to show that (a) branch coverage ensures statement coverage, and (b) statement coverage does not ensure branch coverage.

(a) Branch testing would guarantee statement coverage since every statement must belong to some branch (assuming that there is no unreachable code).

(b) To show that statement coverage does not ensure branch coverage, it would be sufficient to give an example of a test suite that achieves statement coverage, but does not cover at least one branch. Consider the following code, and the test suite {5}.

```
if(x>2) x+=1;
```

The test suite would achieve statement coverage. However, it does not achieve branch coverage, since the condition (x > 2) is not made false by any test case in the suite.

10.7.4 Multiple Condition Coverage

In the multiple condition (MC) coverage-based testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values. For example, consider the composite conditional expression ((c1 .and.c2).or.c3). A test suite would achieve MC coverage, if all the component conditions c1, c2 and c3 are each made to assume both true and false values. Branch testing can be considered to be a simplistic condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values. It is easy to prove that condition testing is a stronger testing strategy than branch testing. For a composite conditional expression of n components, 2n test cases are required for multiple condition coverage. Thus, for multiple condition coverage, the number of test cases increases exponentially with the number of component conditions. Therefore, multiple condition coverage-based testing technique is practical only if n (the number of conditions) is small.

Example 10.13 Give an example of a fault that is detected by multiple condition coverage, but not by branch coverage.

Answer: Consider the following C program segment:

```
if(temperature>150 || temperature>50)
  setWarningLightOn();
```

The program segment has a bug in the second component condition, it should have been `temperature<50`. The test suite {temperature=160, temperature=40} achieves branch coverage. But, it is not able to check that `setWarningLightOn();` should not be called for temperature values within 150 and 50.

10.7.5 Path Coverage

A test suite achieves path coverage if it exeutes each linearly independent paths (o r basis paths) at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program. Therefore, to understand path coverage-based testing strategy, we need to first understand how the CFG of a program can be drawn.

Control flow graph (CFG)

A control flow graph describes how the control flows through the program. We can define a control flow graph as the following:

> A control flow graph describes the sequence in which the different instructions of a program get executed.

In order to draw the control flow graph of a program, we need to first number all the statements of a program. The different numbered statements serve as nodes of the control flow graph (see Figure 10.5). There exists an edge from one node to another, if the execution of the statement representing the first node can result in the transfer of control to the other node.

More formally, we can define a CFG as follows. A CFG is a directed graph consisting of a set of nodes and edges (N, E), such that each node n □ N corresponds to a unique program statement and an edge exists between two nodes if control can transfer from one node to the other.

We can easily draw the CFG for any program, if we know how to represent the sequence, selection, and iteration types of statements in the CFG. After all, every program is constructed by using these three types of constructs only. Figure 10.5 summarises how the CFG for these three types of constructs can be drawn. The CFG representation of the sequence and decision types of statements is straight forward. Please note carefully how the CFG for the loop

(iteration) construct can be drawn. For iteration type of constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore always control flows from the last statement of the loop to the top of the loop. That is, the loop construct terminates from the first statement (after the loop is found to be false) and does not at any time exit the loop at the last statement of the loop. Using these basic ideas, the CFG of the program given in Figure 10.7(a) can be drawn as shown in Figure 10.7(b).

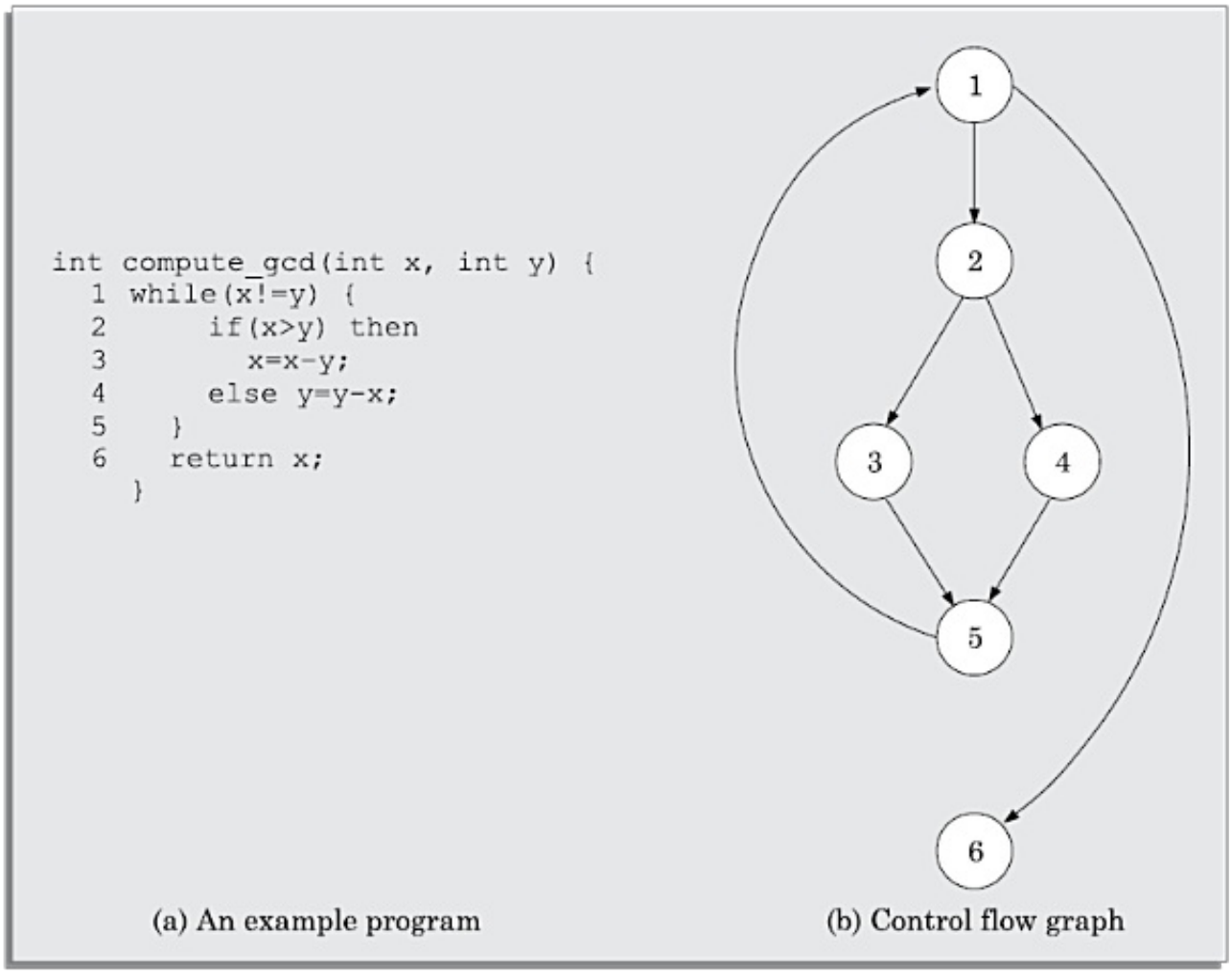

Figure 10.7: Control flow diagram of an example program.

Path

A path through a program is any node and edge sequence from the start node to a terminal node of the control flow graph of a program. Please note that a program can have more than one terminal nodes when it contains multiple exit or return type of statements. Writing test cases to cover all paths of a typical program is impractical since there can be an infinite number of paths through a program in presence of loops. For example, in Figure 10.5(c), there can be an infinite number of paths

such as 12314, 12312314, 12312312314, etc. If coverage of all paths is attempted, then the number of test cases required would become infinitely large. For this reason, path coverage testing does not try to cover all paths, but only a subset of paths called linearly independent paths (or basis paths). Let us now discuss what are linearly independent paths and how to determine these in a program.

Linearly independent set of paths (or basis path set)

A set of paths for a given program is called linearly independent set of paths (or the set of basis paths or simply the basis set), if each path in the set introduces at least one new edge that is not included in any other path in the set. Please note that even if we find that a path has one new node compared to all other linearly independent paths, then this path should also be included in the set of linearly independent paths. This is because, any path having a new node would automatically have a new edge. An alternative definition of a linearly independent set of paths [McCabe76] is the following:

> If a set of paths is linearly independent of each other, then no path in the set can be obtained through any linear operations (i.e., additions or subtractions) on the other paths in the set.

According to the above definition of a linearly independent set of paths, for any path in the set, its subpath cannot be a member of the set. In fact, any arbitrary path of a program, can be synthesized by carrying out linear operations on the basis paths. Possibly, the name basis set comes from the observation that the paths in the basis set form the "basis" for all the paths of a program. Please note that there may not always exist a unique basis set for a program and several basis sets for the same program can usually be determined.

Even though it is straight forward to identify the linearly independent paths for simple programs, for more complex programs it is not easy to determine the number of independent paths. In this context, McCabe's cyclomatic complexity metric is an important result that lets us compute the number of linearly independent paths for any arbitrary program. McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program. Also, the McCabe's cyclomatic complexity is very simple to compute. Though the McCabe's metric does not directly identify the linearly independent paths, but it provides us with a practical way of determining approximately how many paths to look for.

10.7.6 McCabe's Cyclomatic Complexity Metric

McCabe obtained his results by applying graph-theoretic techniques to the control flow graph ofa program. McCabe's cyclomatic complexity defines an upper bound on the number of independent paths in a program. We discuss three different ways to compute the cyclomatic complexity. For structured programs, the results computed by all the three methods are guaranteed to agree.

Method 1: Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as:

$$V(G) = E - N + 2$$

where, N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

For the CFG of example shown in Figure 10.7, E = 7 and N = 6. Therefore, the value of the Cyclomatic complexity = 7 – 6 + 2 = 3.

Method 2: An alternate way of computing the cyclomatic complexity of a program is based on a visual inspection of the control flow graph is as follows —In this method, the cyclomatic complexity V (G) for a graph G is given by the following expression:

```
V(G) = Total number of non-overlapping bounded areas + 1
```

In the program's control flow graph G, any region enclosed by nodes and edges can be called as a bounded area. This is an easy way to determine the McCabe's cyclomatic complexity. But, what if the graph G is not planar (i.e., how ever you draw the graph, two or more edges always intersect). Actually, it can be shown that control flow representation of structured programs always yields planar graphs. But, presence of GOTO's can easily add intersecting edges. Therefore, for non-structured programs, this way of computing the McCabe's cyclomatic complexity does not apply.

The number of bounded areas in a CFG increases with the number of decision statements and loops. Therefore, the McCabe's metric provides a quantitative measure of testing difficulty and the ultimate reliability of a program. Consider the CFG example shown in Figure 10.7. From a visual examination of the CFG the number of bounded areas is 2. Therefore the cyclomatic complexity, computed with this method is also 2+1=3. This method provides a very easy way of computing the cyclomatic complexity of CFGs, just from a visual examination of the CFG. On the other hand, the method for computing CFGs can easily be automated. That is, the McCabe's metric computations methods 1 and 3 can be easily coded into a program

that can be used to automatically determine the cyclomatic complexities of arbitrary programs.

Method 3: The cyclomatic complexity of a program can also be easily computed by computing the number of decision and loop statements of the program. If N is the number of decision and loop statements of a program, then the McCabe's metric is equal to N + 1.

How is path testing carried out by using computed McCabe's cyclomatic metric value?

Knowing the number of basis paths in a program does not make it any easier to design test cases for path coverage, only it gives an indication of the minimum number of test cases required for path coverage. For the CFG of a moderately complex program segment of say 20 nodes and 25 edges, you may need several days of effort to identify all the linearly independent paths in it and to design the test cases. It is therefore impractical to require the test designers to identify all the linearly independent paths in a code, and then design the test cases to force execution along each of the identified paths. In practice, for path testing, usually the tester keeps on forming test cases with random data and executes those until the required coverage is achieved. A testing tool such as a dynamic program analyser (see Section 10.8.2) is used to determine the percentage of linearly independent paths covered by the test cases that have been executed so far. If the percentage of linearly independent paths covered is below 90 per cent, more test cases (with random inputs) are added to increase the path coverage. Normally, it is not practical to target achievement of 100 per cent path coverage, since, the McCabe's metric is only an upper bound and does not give the exact number of paths.

Steps to carry out path coverage-based testing

The following is the sequence of steps that need to be undertaken for deriving the path coverage-based test cases for a program:

1. Draw control flow graph for the program.
2. Determine the McCabe's metric V(G).
3. Determine the cyclomatic complexity. This gives the minimum number of test cases required to achieve path coverage.
4. repeat

Test using a randomly designed set of test cases.
Perform dynamic analysis to check the path coverage achieved.
until at least 90 per cent path coverage is achieved.

Uses of McCabe's cyclomatic complexity metric

Beside its use in path testing, cyclomatic complexity of programs has many other interesting applications such as the following:

Estimation of structural complexity of code: McCabe's cyclomatic complexity is a measure of the structural complexity of a program. The reason for this is that it is computed based on the code structure (number of decision and iteration constructs used). Intuitively, the McCabe's complexity metric correlates with the difficulty level of understanding a program, since one understands a program by understanding the computations carried out along all independent paths of the program.

Cyclomatic complexity of a program is a measure of the psychological complexity or the level of difficulty in understanding the program.

In view of the above result, from the maintenance perspective, it makes good sense to limit the cyclomatic complexity of the different functions to some reasonable value. Good software development organisations usually restrict the cyclomatic complexity of different functions to a maximum value of ten or so. This is in contrast to the computational complexity that is based on the execution of the program statements.

Estimation of testing effort: Cyclomatic complexity is a measure of the maximum number of basis paths. Thus, it indicates the minimum number of test cases required to achieve path coverage. Therefore, the testing effort and the time required to test a piece of code satisfactorily is proportional to the cyclomatic complexity of the code. To reduce testing effort, it is necessary to restrict the cyclomatic complexity of every function to seven.

Estimation of program reliability: Experimental studies indicate there exists a clear relationship between the McCabe's metric and the number of errors latent in the code after testing. This relationship exists possibly due to the correlation of cyclomatic complexity with the structural complexity of code. Usually the larger is the structural complexity, the more difficult it is to test and debug the code.

10.7.7 Data Flow-based Testing

Data flow based testing method selects test paths of a program

according to the definitions and uses of different variables in a program. Consider a program P . For a statement numbered S of P , let

DEF(S) = {X /statement S contains a definition of X } and

USES(S)= {X /statement S contains a use of X }

For the statement S: a=b+c;, DEF(S)={a}, USES(S)={b, c}. The definition of variable X at statement S is said to be live at statement S1 , if there exists a path from statement S to statement S1 which does not contain any definition of X .

All definitions criterion is a test coverage criterion that requires that an adequate test set should cover all definition occurrences in the sense that, for each definition occurrence, the testing paths should cover a path through which the definition reaches a use of the definition. All use criterion requires that all uses of a definition should be covered. Clearly, all-uses criterion is stronger than all-definitions criterion. An even stronger criterion is all definition-use-paths criterion, which requires the coverage of all possible definition-use paths that either are cycle-free or have only simple cycles. A simple cycle is a path in which only the end node and the start node are the same.

The definition-use chain (or DU chain) of a variable X is of the form [X, S, S1], where S and S1 are statement numbers, such that X □ DEF(S) and X □ USES(S1), and the definition of X in the statement S is live at statement S1 . One simple data flow testing strategy is to require that every DU chain be covered at least once. Data flow testing strategies are especially useful for testing programs containing nested if and loop statements.

10.7.8 Mutation Testing

All white-box testing strategies that we have discussed so far, are coverage-based testing techniques. In contrast, mutation testing is a fault-based testing technique in the sense that mutation test cases are designed to help detect specific types of faults in a program. In mutation testing, a program is first tested by using an initial test suite designed by using various white box testing strategies that we have discussed. After the initial testing is complete, mutation testing can be taken up.

The idea behind mutation testing is to make a few arbitrary changes to a program at a time. Each time the program is changed, it is called a mutated program and the change effected is called a mutant. An underlying assumption behind mutation testing is that all programming errors can be

expressed as a combination of simple errors. A mutation operator makes specific changes to a program. For example, one mutation operator may randomly delete a program statement. A mutant may or may not cause an error in the program. If a mutant does not introduce any error in the program, then the original program and the mutated program are called equivalent programs.

A mutated program is tested against the original test suite of the program. If there exists at least one test case in the test suite for which a mutated program yields an incorrect result, then the mutant is said to be dead, since the error introduced by the mutation operator has successfully been detected by the test suite. If a mutant remains alive even after all the test cases have been exhausted, the test suite is enhanced to kill the mutant. However, it is not this straightforward. Remember that there is a possibility of a mutated program to be an equivalent program. When this is the case, it is futile to try to design a test case that would identify the error.

An important advantage of mutation testing is that it can be automated to a great extent. The process of generation of mutants can be automated by predefining a set of primitive changes that can be applied to the program. These primitive changes can be simple program alterations such as—deleting a statement, deleting a variable definition, changing the type of an arithmetic operator (e.g., + to -), changing a logical operator (`and` to `or`) changing the value of a constant, changing the data type of a variable, etc. A major pitfall of the mutation-based testing approach is that it is computationally very expensive, since a large number of possible mutants can be generated.

Mutation testing involves generating a large number of mutants. Also each mutant needs to be tested with the full test suite. Obviously therefore, mutation testing is not suitable for manual testing. Mutation testing is most suitable to be used in conjunction of some testing tool that should automatically generate the mutants and run the test suite automatically on each mutant. At present, several test tools are available that automatically generate mutants for a given program.

10.8 DEBUGGING

After a failure has been detected, it is necessary to first identify the program statement(s) that are in error and are responsible for the failure, the error can then be fixed. In this Section, we shall summarise the important approaches that are available to identify the error locations. Each of these approaches has its own advantages and

disadvantages and therefore each will be useful in appropriate circumstances. We also provide some guidelines for effective debugging.

10.8.1 Debugging Approaches

The following are some of the approaches that are popularly adopted by the programmers for debugging:

Brute force method

This is the most common method of debugging but is the least efficient method. In this approach, print statements are inserted throughout the program to print the intermediate values with the hope that some of the printed values will help to identify the statement in error. This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger), because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly. Single stepping using a symbolic debugger is another form of this approach, where the developer mentally computes the expected result after every source instruction and checks whether the same is computed by single stepping through the program.

Backtracking

This is also a fairly common approach. In this approach, starting from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered. Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases and may become unmanageably large for complex programs, limiting the use of this approach.

Cause elimination method

In this approach, once a failure is observed, the symptoms of the failure (i.e., certain variable is having a negative value though it should be positive, etc.) are noted. Based on the failure symptoms, the causes which could possibly have contributed to the symptom is developed and tests are conducted to eliminate each. A related technique of identification of the error from the error symptom is the software fault tree analysis.

Program slicing

This technique is similar to back tracking. In the backtracking approach, one often has to examine a large number of statements. However, the search space is reduced by defining slices. A slice of a program for a particular variable and at a particular statement is the set of source lines preceding this statement that can influence the value of that variable [Mund2002]. Program slicing makes use of the fact that an error in the value of a variable can be caused by the statements on which it is data dependent.

10.8.2 Debugging Guidelines

Debugging is often carried out by programmers based on their ingenuity and experience. The following are some general guidelines for effective debugging:

- Many times debugging requires a thorough understanding of the program design. Trying to debug based on a partial understanding of the program design may require an inordinate amount of effort to be put into debugging even for simple problems.
- Debugging may sometimes even require full redesign of the system. In such cases, a common mistakes that novice programmers often make is attempting not to fix the error but its symptoms.
- One must be beware of the possibility that an error correction may introduce new errors. Therefore after every round of error-fixing, regression testing (see Section 10.13) must be carried out.

10.9 PROGRAM ANALYSIS TOOLS

A program analysis tool usually is an automated tool that takes either the source code or the executable code of a program as input and produces reports regarding several important characteristics of the program, such as its size, complexity, adequacy of commenting, adherence to programming standards, adequacy of testing, etc. We can classify various program analysis tools into the following two broad categories:

- Static analysis tools
- Dynamic analysis tools

These two categories of program analysis tools are discussed in the following subsection.

10.9.1 Static Analysis Tools

Static program analysis tools assess and compute various characteristics of a program without executing it. Typically, static analysis tools analyse the source code to compute certain metrics characterising the source code (such as size, cyclomatic complexity, etc.) and also report certain analytical conclusions. These also check the conformance of the code with the prescribed coding standards. In this context, it displays the following analysis results:

- To what extent the coding standards have been adhered to?
- Whether certain programming errors such as uninitialised variables, mismatch between actual and formal parameters, variables that are declared but never used, etc., exist? A list of all such errors is displayed.

Code review techniques such as code walkthrough and code inspection discussed in Sections 10.2.1 and 10.2.2 can be considered as static analysis methods since those target to detect errors based on analysing the source code. However, strictly speaking, this is not true since we are using the term static program analysis to denote automated analysis tools. On the other hand, a compiler can be considered to be a type of a static program analysis tool.

A major practical limitation of the static analysis tools lies in their inability to analyse run-time information such as dynamic memory references using pointer variables and pointer arithmetic, etc. In a high level programming languages, pointer variables and dynamic memory allocation provide the capability for dynamic memory references. However, dynamic memory referencing is a major source of programming errors in a program.

Static analysis tools often summarise the results of analysis of every function in a polar chart known as Kiviat Chart. A Kiviat Chart typically shows the analysed values for cyclomatic complexity, number of source lines, percentage of comment lines, Halstead's metrics, etc.

10.9.2 Dynamic Analysis Tools

Dynamic program analysis tools can be used to evaluate several program

characteristics based on an analysis of the run time behaviour of a program. These tools usually record and analyse the actual behaviour of a program while it is being executed. A dynamic program analysis tool (also called a dynamic analyser) usually collects execution trace information by instrumenting the code. Code instrumentation is usually achieved by inserting additional statements to print the values of certain variables into a file to collect the execution trace of the program. The instrumented code when executed, records the behaviour of the software for different test cases.

An important characteristic of a test suite that is computed by a dynamic analysis tool is the extent of coverage achieved by the test suite.

After a software has been tested with its full test suite and its behaviour recorded, the dynamic analysis tool carries out a post execution analysis and produces reports which describe the coverage that has been achieved by the complete test suite for the program. For example, the dynamic analysis tool can report the statement, branch, and path coverage achieved by a test suite. If the coverage achieved is not satisfactory more test cases can be designed, added to the test suite, and run. Further, dynamic analysis results can help eliminate redundant test cases from a test suite.

Normally the dynamic analysis results are reported in the form of a histogram or pie chart to describe the structural coverage achieved for different modules of the program. The output of a dynamic analysis tool can be stored and printed easily to provide evidence that thorough testing has been carried out.

10.10 INTEGRATION TESTING

Integration testing is carried out after all (or at least some of) the modules have been unit tested. Successful completion of unit testing, to a large extent, ensures that the unit (or module) as a whole works satisfactorily. In this context, the objective of integration testing is to detect the errors at the module interfaces (call parameters). For example, it is checked that no parameter mismatch occurs when one module invokes the functionality of another module. Thus, the primary objective of integration testing is to test the module interfaces, i.e., there are no errors in parameter passing, when one module invokes the functionality of another module.

The objective of integration testing is to check whether the different modules of a program interface with each other properly.

During integration testing, different modules of a system are integrated in a planned manner using an integration plan. The integration plan specifies the steps and the order in which modules are combined to realise the full system. After each integration step, the partially integrated system is tested.

An important factor that guides the integration plan is the module dependency graph.

We have already discussed in Chapter 6 that a structure chart (or module dependency graph) specifies the order in which different modules call each other. Thus, by examining the structure chart, the integration plan can be developed. Any one (or a mixture) of the following approaches can be used to develop the test plan:

- Big-bang approach to integration testing
- Top-down approach to integration testing
- Bottom-up approach to integration testing
- Mixed (also called sandwiched) approach to integration testing

In the following subsections, we provide an overview of these approaches to integration testing.

Big-bang approach to integration testing

Big-bang testing is the most obvious approach to integration testing. In this approach, all the modules making up a system are integrated in a single step. In simple words, all the unit tested modules of the system are simply linked together and tested. However, this technique can meaningfully be used only for very small systems. The main problem with this approach is that once a failure has been detected during integration testing, it is very difficult to localise the error as the error may potentially lie in any of the modules. Therefore, debugging errors reported during big-bang integration testing are very expensive to fix. As a result, big-bang integration testing is almost never used for large programs.

Bottom-up approach to integration testing

Large software products are often made up of several subsystems. A subsystem might consist of many modules which communicate among each other through well-defined interfaces. In bottom-up integration testing, first the modules for the each subsystem are integrated. Thus, the subsystems can be integrated separately and independently.

The primary purpose of carrying out the integration testing a subsystem is to test whether the interfaces among various modules making up the subsystem work satisfactorily. The test cases must be carefully chosen to exercise the interfaces in all possible manners.

In a pure bottom-up testing no stubs are required, and only test-drivers are required. Large software systems normally require several levels of subsystem testing, lower-level subsystems are successively combined to form higher-level subsystems. The principal advantage of bottom- up integration testing is that several disjoint subsystems can be tested simultaneously. Another advantage of bottom-up testing is that the low-level modules get tested thoroughly, since they are exercised in each integration step. Since the low-level modules do I/O and other critical functions, testing the low-level modules thoroughly increases the reliability of the system. A disadvantage of bottom-up testing is the complexity that occurs when the system is made up of a large number of small subsystems that are at the same level. This extreme case corresponds to the big-bang approach.

Top-down approach to integration testing

Top-down integration testing starts with the root module in the structure chart and one or two subordinate modules of the root module. After the top-level 'skeleton' has been tested, the modules that are at the immediately lower layer of the 'skeleton' are combined with it and tested. Top-down integration testing approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test. A pure top-down integration does not require any driver routines. An advantage of top-down integration testing is that it requires writing only stubs, and stubs are simpler to write compared to drivers. A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, it becomes difficult to exercise the top-level routines in the desired manner since the lower level routines usually perform input/output (I/O) operations.

Mixed approach to integration testing

The mixed (also called sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level modules have been coded and unit tested. Similarly, bottom-up testing can start only

after the bottom level modules are ready. The mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. In the mixed testing approach, testing can start as and when modules become available after unit testing. Therefore, this is one of the most commonly used integration testing approaches. In this approach, both stubs and drivers are required to be designed.

10.10.1 Phased versus Incremental Integration Testing

Big-bang integration testing is carried out in a single step of integration. In contrast, in the other strategies, integration is carried out over several steps. In these later strategies, modules can be integrated either in a phased or incremental manner. A comparison of these two strategies is as follows:

- In incremental integration testing, only one new module is added to the partially integrated system each time.
- In phased integration, a group of related modules are added to the partial system each time.

Obviously, phased integration requires less number of integration steps compared to the incremental integration approach. However, when failures are detected, it is easier to debug the system while using the incremental testing approach since the errors can easily be traced to the interface of the recently integrated module. Please observe that a degenerate case of the phased integration testing approach is big-bang testing.

10.11 TESTING OBJECT-ORIENTED PROGRAMS

During the initial years of object-oriented programming, it was believed that object-orientation would, to a great extent, reduce the cost and effort incurred on testing. This thinking was based on the observation that object-orientation incorporates several good programming features such as encapsulation, abstraction, reuse through inheritance, polymorphism, etc., thereby chances of errors in the code is minimised. However, it was soon realised that satisfactory testing object-oriented programs is much more difficult and requires much more cost and effort as compared to testing similar procedural programs. The main reason behind this situation is that various object-oriented features introduce additional complications and scope of new types of bugs that are

present in procedural programs. Therefore additional test cases are needed to be designed to detect these. We examine these issues as well as some other basic issues in testing object-oriented programs in the following subsections.

10.11.1 What is a Suitable Unit for Testing Object-oriented Programs?

For procedural programs, we had seen that procedures are the basic units of testing. That is, first all the procedures are unit tested. Then various tested procedures are integrated together and tested. Thus, as far as procedural programs are concerned, procedures are the basic units of testing. Since methods in an object-oriented program are analogous to procedures in a procedural program, can we then consider the methods of object-oriented programs as the basic unit of testing? Weyuker studied this issue and postulated his anticomposition axiom as follows:

> Adequate testing of individual methods does not ensure that a class has been satisfactorily tested.

The main intuitive justification for the anticomposition axiom is the following. A method operates in the scope of the data and other methods of its object. That is, all the methods share the data of the class. Therefore, it is necessary to test a method in the context of these. Moreover, objects can have significant number of states. The behaviour of a method can be different based on the state of the corresponding object. Therefore, it is not enough to test all the methods and check whether they can be integrated satisfactorily. A method has to be tested with all the other methods and data of the corresponding object. Moreover, a method needs to be tested at all the states that the object can assume. As a result, it is improper to consider a method as the basic unit of testing an object-oriented program.

> An object is the basic unit of testing of object-oriented programs.

Thus, in an object oriented program, unit testing would mean testing each object in isolation. During integration testing (called cluster testing in the object-oriented testing literature) various unit tested objects are integrated and tested. Finally, system-level testing is carried out.

10.11.2 Do Various Ob ject-orientation Features Make Testing Easy?

In this section, we discuss the implications of different object-orientation features in testing.

Encapsulation: We had discussed in Chapter 7 that the encapsulation feature helps in data abstraction, error isolation, and error prevention. However, as far as testing is concerned, encapsulation is not an obstacle to testing, but leads to difficulty during debugging. Encapsulation prevents the tester from accessing the data internal to an object. Of course, it is possible that one can require classes to support state reporting methods to print out all the data internal to an object. Thus, the encapsulation feature though makes testing difficult, the difficulty can be overcome to some extent through use of appropriate state reporting methods.

Inheritance: The inheritance feature helps in code reuse and was expected to simplify testing. It was expected that if a class is tested thoroughly, then the classes that are derived from this class would need only incremental testing of the added features. However, this is not the case.

Even if the base class class has been thoroughly tested, the methods inherited from the base class need to be tested again in the derived class.

The reason for this is that the inherited methods would work in a new context (new data and method definitions). As a result, correct behaviour of a method at an upper level, does not guarantee correct behaviour at a lower level. Therefore, retesting of inherited methods needs to be followed as a rule, rather as an exception.

Dynamic binding: Dynamic binding was introduced to make the code compact, elegant, and easily extensible. However, as far as testing is concerned all possible bindings of a method call have to be identified and tested. This is not easy since the bindings take place at run-time.

Object states: In contrast to the procedures in a procedural program, objects store data permanently. As a result, objects do have significant states. The behaviour of an object is usually different in different states. That is, some methods may not be active in some of its states. Also, a method may act differently in different states. For example, when a book has been issued out in a library information system, the book reaches the issuedOut state. In this state, if the issue method is invoked, then it may not exhibit its normal behaviour.

In view of the discussions above, testing an object in only one of its states is not enough. The object has to be tested at all its possible states. Also,

whether all the transitions between states (as specified in the object model) function properly or not should be tested. Additionally, it needs to be tested that no extra (sneak) transitions exist, neither are there extra states present other than those defined in the state model. For state-based testing, it is therefore beneficial to have the state model of the objects, so that the conformance of the object to its state model can be tested.

10.11.3 Why are Traditional Techniques Considered Not Satisfactory for Testing Object-oriented Programs?

We have already seen that in traditional procedural programs, procedures are the basic unit of testing. In contrast, objects are the basic unit of testing for object-oriented programs. Besides this, there are many other significant differences as well between testing procedural and object-oriented programs. For example, statement coverage-based testing which is popular for testing procedural programs is not meaningful for object-oriented programs. The reason is that inherited methods have to be retested in the derived class. In fact, the different object- oriented features (inheritance, polymorphism, dynamic binding, state-based behaviour, etc.) require special test cases to be designed compared to the traditional testing as discussed in Section 10.11.4. The various object-orientation features are explicit in the design models, and it is usually difficult to extract from and analysis of the source code. As a result, the design model is a valuable artifact for testing object-oriented programs. Test cases are designed based on the design model. Therefore, this approach is considered to be intermediate between a fully white-box and a fully black-box approach, and is called a grey-box approach. Please note that grey-box testing is considered important for object-oriented programs. This is in contrast to testing procedural programs.

10.11.4 Grey-Box Testing of Object-oriented Programs

As we have already mentioned, model-based testing is important for object-oriented programs, as these test cases help detect bugs that are specific to the object-orientation constructs.

For object-oriented programs, several types of test cases can be designed based on the design models of object-oriented programs. These are called the grey-box test cases.

The following are some important types of grey-box testing that can be carried on based on UML models:

State-model-based testing

State coverage: Each method of an object are tested at each state of the object.

State transition coverage: It is tested whether all transitions depicted in the state model work satisfactorily.

State transition path coverage: All transition paths in the state model are tested.

Use case-based testing

Scenario coverage: Each use case typically consists of a mainline scenario and several alternate scenarios. For each use case, the mainline and all alternate sequences are tested to check if any errors show up.

Class diagram-based testing

Testing derived classes: All derived classes of the base class have to be instantiated and tested. In addition to testing the new methods defined in the derivec. lass, the inherited methods must be retested.

Association testing: All association relations are tested.

Aggregation testing: Various aggregate objects are created and tested.

Sequence diagram-based testing

Method coverage: All methods depicted in the sequence diagrams are covered. **Message path coverage:** All message paths that can be constructed from the sequence diagrams are covered.

10.11.5 Integration Testing of Object-oriented Programs

There are two main approaches to integration testing of object-oriented programs:

- Thread-based
- Use based

Thread-based approach: In this approach, all classes that need to collaborate to realise the behaviour of a single use case are integrated and tested. After all the required classes for a use case are integrated and tested,

another use case is taken up and other classes (if any) necessary for execution of the second use case to run are integrated and tested. This is continued till all use cases have been considered.

Use-based approach: Use-based integration begins by testing classes that either need no service from other classes or need services from at most a few other classes. After these classes have been integrated and tested, classes that use the services from the already integrated classes are integrated and tested. This is continued till all the classes have been integrated and tested.

10.12 SYSTEM TESTING

After all the units of a program have been integrated together and tested, system testing is taken up.

System tests are designed to validate a fully developed system to assure that it meets its requirements. The test cases are therefore designed solely based on the SRS document.

The system testing procedures are the same for both object-oriented and procedural programs, since system test cases are designed solely based on the SRS document and the actual implementation (procedural or object-oriented) is immaterial.

There are essentially three main kinds of system testing depending on who carries out testing:

1. **Alpha Testing:** Alpha testing refers to the system testing carried out by the test team within the developing organisation.
2. **Beta Testing:** Beta testing is the system testing performed by a select group of friendly customers.
3. **Acceptance Testing:** Acceptance testing is the system testing performed by the customer to determine whether to accept the delivery of the system.

In each of the above types of system tests, the test cases can be the same, but the difference is with respect to who designs test cases and carries out testing.

The system test cases can be classified into functionality and performance test cases.

Before a fully integrated system is accepted for system testing, smoke testing is performed. Smoke testing is done to check whether at least the

main functionalities of the software are working properly. Unless the software is stable and at least the main functionalities are working satisfactorily, system testing is not undertaken.

The functionality tests are designed to check whether the software satisfies the functional requirements as documented in the SRS document. The performance tests, on the other hand, test the conformance of the system with the non-functional requirements of the system. We have already discussed how to design the functionality test cases by using a black-box approach (in Section 10.5 in the context of unit testing). So, in the following subsection we discuss only smoke and performance testing.

10.12.1 Smoke Testing

Smoke testing is carried out before initiating system testing to ensure that system testing would be meaningful, or whether many parts of the software would fail. The idea behind smoke testing is that if the integrated program cannot pass even the basic tests, it is not ready for a vigorous testing. For smoke testing, a few test cases are designed to check whether the basic functionalities are working. For example, for a library automation system, the smoke tests may check whether books can be created and deleted, whether member records can be created and deleted, and whether books can be loaned and returned.

10.12.2 Performance Testing

Performance testing is an important type of system testing.

> Performance testing is carried out to check whether the system meets the non-functional requirements identified in the SRS document.

There are several types of performance testing corresponding to various types of non-functional requirements. For a specific system, the types of performance testing to be carried out on a system depends on the different non-functional requirements of the system documented in its SRS document. All performance tests can be considered as black-box tests.

Stress testing

Stress testing is also known as endurance testing. Stress testing evaluates system performance when it is stressed for short periods of time. Stress tests are black-box tests which are designed to impose a range of abnormal and even illegal input conditions so as to stress the

capabilities of the software. Input data volume, input data rate, processing time, utilisation of memory, etc., are tested beyond the designed capacity. For example, suppose an operating system is supposed to support fifteen concurrent transactions, then the system is stressed by attempting to initiate fifteen or more transactions simultaneously. A real-time system might be tested to determine the effect of simultaneous arrival of several high-priority interrupts.

Stress testing is especially important for systems that under normal circumstances operate below their maximum capacity but may be severely stressed at some peak demand hours. For example, if the corresponding non-functional requirement states that the response time should not be more than twenty secs per transaction when sixty concurrent users are working, then during stress testing the response time is checked with exactly sixty users working simultaneously.

Volume testing

Volume testing checks whether the data structures (buffers, arrays, queues, stacks, etc.) have been designed to successfully handle extraordinary situations. For example, the volume testing for a compiler might be to check whether the symbol table overflows when a very large program is compiled.

Configuration testing

Configuration testing is used to test system behaviour in various hardware and software configurations specified in the requirements. Sometimes systems are built to work in different configurations for different users. For instance, a minimal system might be required to serve a single user, and other extended configurations may be required to serve additional users during configuration testing. The system is configured in each of the required configurations and depending on the specific customer requirements, it is checked if the system behaves correctly in all required configurations.

Compatibility testing

This type of testing is required when the system interfaces with external systems (e.g., databases, servers, etc.). Compatibility aims to check whether the interfaces with the external systems are performing as required. For instance, if the system needs to communicate with a large

database system to retrieve information, compatibility testing is required to test the speed and accuracy of data retrieval.

Regression testing

This type of testing is required when a software is maintained to fix some bugs or enhance functionality, performance, etc. Regression testing is also discussed in Section 10.13.

Recovery testing

Recovery testing tests the response of the system to the presence of faults, or loss of power, devices, services, data, etc. The system is subjected to the loss of the mentioned resources (as discussed in the SRS document) and it is checked if the system recovers satisfactorily. For example, the printer can be disconnected to check if the system hangs. Or, the power may be shut down to check the extent of data loss and corruption.

Maintenance testing

This addresses testing the diagnostic programs, and other procedures that are required to help maintenance of the system. It is verified that the artifacts exist and they perform properly.

Documentation testing

It is checked whether the required user manual, maintenance manuals, and technical manuals exist and are consistent. If the requirements specify the types of audience for which a specific manual should be designed, then the manual is checked for compliance of this requirement.

Usability testing

Usability testing concerns checking the user interface to see if it meets all user requirements concerning the user interface. During usability testing, the display screens, messages, report formats, and other aspects relating to the user interface requirements are tested. A GUI being just being functionally correct is not enough. Therefore, the GUI has to be checked against the checklist we discussed in Sec. 9.5.6.

Security testing

Security testing is essential for software that handle or process confidential data that is to be gurarded against pilfering. It needs to be tested whether the system is fool-proof from security attacks such as intrusion by hackers. Over the last few years, a large number of security testing techniques have been proposed, and these include password cracking, penetration testing, and attacks on specific ports, etc.

10.12.3 Error Seeding

Sometimes customers specify the maximum number of residual errors that can be present in the delivered software. These requirements are often expressed in terms of maximum number of allowable errors per line of source code. The error seeding technique can be used to estimate the number of residual errors in a software.

Error seeding, as the name implies, it involves seeding the code with some known errors. In other words, some artificial errors are introduced (seeded) into the program. The number of these seeded errors that are detected in the course of standard testing is determined. These values in conjunction with the number of unseeded errors detected during testing can be used to predict the following aspects of a program:

- The number of errors remaining in the product.
- The effectiveness of the testing strategy.

Let N be the total number of defects in the system, and let n of these defects be found by testing.

Let S be the total number of seeded defects, and let s of these defects be found during testing. Therefore, we get:

$$\frac{n}{N} = \frac{s}{S}$$

or

$$N = S \times \frac{n}{s}$$

Defects still remaining in the program after testing can be given by:

$$N - n = n \times \frac{(S-1)}{s}$$

Error seeding works satisfactorily only if the kind seeded errors and their frequency of occurrence matches closely with the kind of defects that actually exist. However, it is difficult to predict the types of errors that exist in a software. To some extent, the different categories of errors that are latent

and their frequency of occurrence can be estimated by analyzing historical data collected from similar projects. That is, the data collected is regarding the types and the frequency of latent errors for all earlier related projects. This gives an indication of the types (and the frequency) of errors that are likely to have been committed in the program under consideration. Based on these data, the different types of errors with the required frequency of occurrence can be seeded.

10.13 SOME GENERAL ISSUES ASSOCIATED WITH TESTING

In this section, we shall discuss two general issues associated with testing. These are—how to document the results of testing and how to perform regression testing.

Test documentation

A piece of documentation that is produced towards the end of testing is the test summary report. This report normally covers each subsystem and represents a summary of tests which have been applied to the subsystem and their outcome. It normally specifies the following:

- What is the total number of tests that were applied to a subsystem.
- Out of the total number of tests how many tests were successful.
- How many were unsuccessful, and the degree to which they were unsuccessful, e.g., whether a test was an outright failure or whether some of the expected results of the test were actually observed.

Regression testing

Regression testing spans unit, integration, and system testing. Instead, it is a separate dimension to these three forms of testing. Regression testing is the practice of running an old test suite after each change to the system or after each bug fix to ensure that no new bug has been introduced due to the change or the bug fix. However, if only a few statements are changed, then the entire test suite need not be run — only those test cases that test the functions and are likely to be affected by the change need to be run. Whenever a software is changed to either fix a bug, or enhance or remove a feature, regression testing is carried out.

SUMMARY

- In this chapter we discussed the coding and testing phases of the software life cycle.
- Most software development organisations formulate their own coding standards and expect their engineers to adhere to them. On the other hand, coding guidelines serve as general suggestions to programmers regarding good programming styles, but the implementation of the guidelines is left to the discretion to the individual engineers.
- Code review is an efficient way of removing errors as compared to testing, because code review identifies errors whereas testing identifies failures. Therefore, after identifying failures, additional efforts (debugging) must be done to locate and fix the errors.
- Exhaustive testing of almost any non-trivial system is impractical. Also, random selection of test cases is inefficient since many test cases become redundant as they detect the same type of errors. Therefore, we need to design an minimal set of test cases that would expose as many errors as possible.
- There are two well-known approaches to testing—black-box testing and white-box testing. Black box testing is also known as functional testing. Designing test cases for black box testing does not require any knowledge about how the functions have been designed and implemented. On the other hand, white-box testing requires knowledge about internals of the software.
- Object-oriented features complicate the testing process as test cases have to be designed to detect bugs that are associated with these new types of features that are specific to object-orientation programs.
- We discussed some important issues in integration and system testing. We observed that the system test suite is designed based on the SRS document. The two major types of system testing are functionality testing and performance testing. The functionality test cases are designed based on the functional requirements and the performance test cases are design to test the compliance of the system to test the non-functional requirements documented in the SRS document.

EXERCISES

1. For each of the following questions, choose the correct option:
   (a) When is code review performed during software life cycle?
   (i) After unit testing
   (ii) After coding and compiling

(iii) During integration testing
(iv) During system testing

(b) Which one of the following assertions is true?
(i) Code inspection is carried out on tested and debugged code.
(ii) Code inspection and code walkthrough are essentially synonymous.
(iii) Adherence to coding standards are checked during code inspection.
(iv) Code walkthrough makes code inspection redundant.

(c) Identify the synonyms from: error, bug, mistake, failure, and fault:
(i) error, bug, mistake, failure, and fault
(ii) error, bug, failure
(iii) error, bug, fault
(iv) bug, failure, fault

(d) Which one of the followings is not a recognised software testing technique?
(i) Data-flow testing
(ii) Path testing
(iii) Syntax testing
(iv) Decision testing

(e) Unit testing of a software module does NOT require testing which one of the following:
(i) Whether coding standards have been followed.
(ii) Whether the functions of the module are working as per design.
(iii) Whether all arithmetic statements of the module are working properly. (iv) Whether all control statements are working properly.

(f) Which one of the following verification and validation (V and V) activity targets to detect noncompliance to coding standard?
(i) Unit testing
(ii) Code inspection
(iii) Code walk through
(iv) System testing

(g) Code review does not target to detect which of the following types of testing: (i) Algorithmic error
(ii) Syntax error
(iii) Programming error
(iv) Logic error

(h) McCabe's cyclomatic complexity is defined in terms of which of the following? (i) A syntax graph

(ii) A data-flow diagram
(iii) A control flow diagram
(iv) A structure chart

(i) Which one of the followings is true about program verification?
(i) Checks that we are building the right system
(ii) Checks that we are building the system right
(iii) Performed by an independent test team
(iv) Ensures that the developed product is what the user really wants

(j) Which one of the following statements is not an objective of software verification?
(i) Ensuring that product development steps are carried out correctly.
(ii) Ensuring that the correct product has been developed.
(iii) Achieving phase containment of errors.
(iv) Ensuring that the outputs produced at a stage conform to the outputs of the previous phase.

(k) Which of the following does not help in achieving phase containment of errors?
(i) System testing
(ii) Review
(iii) Prototyping
(iv) Simulation

(l) After a program has been modified, which one of the following options characterizes the regression test cases?
(i) All test cases
(ii) Test cases that execute the modified statements
(iii) Test cases that execute the affected or modified statements
(iv) Test cases that execute the unaffected statements

(m) Which of the following is a black-box testing approach?
(i) Path testing
(ii) Boundary value testing
(iii) Mutation testing
(iv) Branch testing

(n) Which of the following is not a software verification technique?
(i) Review
(ii) Simulation
(iii) Unit testing
(iv) Theorem proving
(v) Model checking

(vi) Inspection
(vii) Stress testing

(o) Why is it important to test boundary values while testing a function?
(i) It reduces test costs as boundary values are easily computed by hand.
(ii) Debugging is easier when testing boundary values.
(iii) The correct execution of a function on all boundary values proves that a function is correct.
(iv) In practice, programming the boundary conditions are error prone.

(p) Which of the following can be considered as a program validation technique?
(i) Unit testing
(ii) Integration testing
(iii) Code review
(iv) Acceptance testing

(q) If branch coverage has been achieved on a unit under test, which of the following coverage is implicitly implied?
(i) Path coverage
(ii) Multiple condition coverage
(iii) Statement coverage
(iv) Data flow coverage

(r) Which of the following attributes of a program can be inferred from the cyclomatic complexity of a program?
(i) Computational complexity
(ii) Lines of code (LoC)
(iii) Executable code size
(iv) Understandability

2. For each of the following questions, choose the correct option:

(a) Which of the following statements about cyclomatic complexity metric of a program is FALSE?
(i) It is a measure of the testing difficulty of the program.
(ii) It is a measure of understanding difficulty of the program.
(iii) It is a measure of the linearly independent paths in the program
(iv) It is a measure of the size of the program

(b) Alpha and Beta testing are considered to be which one of the following types of testing?
(i) Regression testing
(ii) Unit testing

(iii) Integration testing
(iv) Acceptance testing

(c) The purpose of error seeding is which one of the followings?
(i) Determine the origin of the bugs
(ii) Plant trojans
(iii) Determine the number of latent bugs
(iv) Plant insidious bugs before delivery to the customer

(d) When in the development cycle is code review carried out?
(i) After coding is complete and before the code is compiled.
(ii) After coding is complete and after the code is compiled.
(iii) After unit testing is over
(iv) After system testing is over

(e) If the condition expression in a conditional statement is composed of n atomic conditions, what is the number of test cases required to achieve multiple condition coverage?
(i) n
(ii) 2n
(iii) 2 × n
(iv) 2 × n + 1

(f) If two code segments have cyclomatic complexities of N1 and N2 respectively, what will be the Cyclomatic complexity of the juxtaposition of the two code segments?
(i) N1 + N2
(ii) N1 + N2 + 1
(iii) N1 + N2 – 1
(iv) N1 □ N2

(g) For a large programe which one of the following integration testing strategy is rarely used:
(i) Big-bang
(ii) Top-down
(iii) Bottom-up
(iv) Mixed

(h) Which one of the followings is true of a pure top-down integration testing process?
(i) Requires only stubs for testing
(ii) Requires only drivers for testing
(iii) Requires both stubs and drivers for testing
(iv) Requires neither stubs nor drivers for testing

(i) Which of the following types of testing is not performed during system testing?
(i) Stress testing
(ii) Functionality testing
(iii) Recovery testing
(iv) White-box testing

(j) The principal aim of code coverage analysis is to evaluate the quality of:
(i) Product
(ii) Test cases
(iii) Coding
(iv) Design

(k) Which one of the following types of program models is normally used to design integration test plan?
(i) CFG
(ii) DFD
(iii) Structure chart
(iv) State chart

(l) Which one of the following software tools will best help you determine whether your test cases are fully exercising your code?
(i) Dynamic analyser
(ii) Static analyser
(iii) Parser
(iv) Profiler

(m) Which one of the following integration teting strategies requires stubs to be designed?
(i) Big-bang
(ii) Top-down
(iii) Botoom-up
(iv) Phased bottom-up

(n) Which one of the following integration teting strategies requires drivers to be designed?
(i) Big-bang
(ii) Top-down
(iii) Bottom-up
(iv) Phased top-down

(o) The purpose of error seeding is which one of the followings?
(i) Determine the root cause of the errors

(ii) Appropriately incorporate Trojans
(iii) Appropriately incorporate Byzantine errors
(iv) Determine the number of latent errors

(p) Which one of the followings is usually the most important consideration while codeing?
(i) Productivity
(ii) Readability
(iii) Brevity
(iv) Use of as less memory space as possible

3. Distinguish between an error and a failure in the context of program testing. Testing detects which of these two? Justify your answer.
4. Would you consider an approach in which the tester tests a program using a large number of random values satisfactory? Explain your answer.
5. What are driver and stub modules in the context of integration and unit testing of a software? Why are the stub and driver modules required?
6. State **TRUE** or **FALSE** of the following assertions. Support your answer with proper reasoning:
(a) The effectiveness of a test suite in detecting errors in a system can be determined by counting the number of test cases in the suite.
(b) Once the McCabe's cyclomatic complexity of a program has been determined, it is very easy to identify all the linearly independent paths of the program.
(c) Use of static and dynamic program analysis tools is an effective substitute for thorough testing.
(d) During code review you detect errors whereas during code testing you detect failures.
(e) A pure top-down integration testing does not require the use of any stub modules.
(f) Adherence to coding standards is checked during the system testing stage.
(g) A program usually does not have one unique set of linearly independent paths.
(h) The minimum number of test cases required for branch coverage-based testing of a program can be greater than those required for path coverage-based testing of the same program.
(i) Branch coverage-based testing is a stronger testing strategy compared to path coverage-based testing.

(j) Out of all types of internal documentation (i.e., provided in the source code), careful commenting is the most useful.
(k) Error and failure are synonymous in software testing terminology.
(l) Development of suitable driver and stub functions are essential for carrying out effective system testing of a product.
(m) System testing can be considered to be a white-box testing of a system.
(n) The main purpose of integration testing is to find errors in the body of functions.
(o) Introduction of additional sequence type of statements in a program cannot increase the cyclomatic complexity of the program.
(p) The terms software verification and software validation are essentially synonyms.
(q) Code walkthrough for a module is normally carried out after unit test is over.
(r) Code walkthrough for a module is normally carried out after the module successfully compiles.
(s) During code walkthrough most of the syntax errors are identified.
(t) Code inspection targets to identify algorithmic errors.
(u) Cyclomatic complexity of a piece of code correlates well with the difficulty of testing the code satisfactorily.
(v) Test coverage analysers are essentially static analysers.
(w) System testing of an object-oriented implementation of a system would be considerably easier than that of a procedural implementation of the same system.
(x) A satisfactory way to test object-oriented programs, is to test all the methods supported by different classes individually and then by performing adequate integration and system testing.
(y) A compiler can be considered as a static program analysis tool.
(z) While verification is concerned with phase containment of errors, the aim of validation is that the final product be error free.

7. State **TRUE** or **FALSE** of the following assertions. Support your answer with proper reasoning:
(a) Unit testing of different modules of a program are carried out during the testing phase.
(b) If a stronger testing has been performed, then a weaker testing need not be carried out.
(c) Static code analysers can easily detect all types of "array index out of

bound" type of errors.

(d) Performance testing is planned based on the functional requirements of the product being tested.

(e) Cleanroom testing approach helps to substantially decrease the overall testing effort.

(f) If a base class is thoroughly tested, then the inherited methods in the derived class need not be tested.

(g) Equivalence class partitioning is a white-box testing strategy.

(h) The big-bang approach is preferred for integration testing of large programs.

(i) Test stubs are simpler to write as compared to test drivers.

(j) Coding guidelines are specific suggestions to the programmers, which they may or may not follow.

(k) Suppose the number of loop and conditional constructs used in a program is n, then the cardinality of its basis path set would be 2*n – 1.

(l) If the black-box testing of a program has been successfully carried out, then the white-box testing can be skipped and vice versa.

(m) Statement coverage is not considered to be a satisfactory testing of a program unit.
Briefly explain the reason behind this. Give an example of a bug, that would not be detected through statement coverage testing.

(n) The system testing procedure would be different depending on whether object- oriented or procedural paradigm has been followed in the program development.

(o) Testing detects failures whereas program inspection identifies errors.

8. What is the difference between black-box testing and white-box testing? Give an example of a bug that is detected by the black-box test suite, but is not detected by the white-box test suite, and vice versa.

9. What is the difference between internal and external documentation? What are the different ways of providing internal documentation? Out of these, which is the most useful?

10. What is meant by structural complexity of a program? Define a metric for measuring the structural complexity of a program. How is structural complexity of a program different from its computational complexity? How is structural complexity useful in program development?

11. Write a C function for searching an integer value from a large sorted sequence of integer values stored in an array of size 100, using the

binary search method.
(a) Build the control flow graph of your binary search function, and hence determine its cyclomatic complexity.
(b) How is cyclomatic metric useful in designing test suite for path coverage? (c) Design a test suite for testing your binary search function.

12. What do you understand by positive and negative test cases? Give one example of each.
13. Given a software and its requirements specification document, explain how would you design the system test suite for the software.
14. What is a coding standard? Identify the problems that might occur if the engineers of an organisation do not adhere to any coding standard?
15. What is the difference between a coding standard and a coding guideline? Why are formulation and use of suitable coding standards and guidelines considered important to a software development organisation? Write down five important coding standards and coding guidelines that you would recommend.
16. What do you understand by coding standard?
When during the development process is the compliance with coding standards is checked?
List two coding standards each for
(i) enhancing readability of the code,
(ii) reuse of the code,
(iii) enhancing code maintainability.
17. What do you understand by testability of a program? Between the programs written by two different programmers to essentially the same programming problem, how can you determine which one is more testable?
18. Discuss different types of code reviews. Explain when and how code review meetings are conducted. Why is code review considered to be a more efficient way to remove errors from code compared to testing?
19. Distinguish between software verification and software validation. Can one be used in place of the other? Justify your answer. In which phase(s) of the iterative waterfall SDLC are the verification and validation activities performed?
20. What are the activities carried out during testing a software? Schematically represent these activities. Which one of these activities

takes the maximum effort?

21. Which one of the following is the strongest structural testing technique —statement coverage-based testing, branch coverage-based testing, or multiple condition coverage-based testing? Justify your answer.
22. Prove that branch coverage-based testing technique is a stronger testing technique compared to a statement coverage-based testing technique.
23. Which is a stronger testing—data flow testing or path testing? Give the reasonings behind your answer.
24. Briefly highlight the difference between code inspection and code walkthrough. Compare the relative merits of code inspection and code walkthrough.
25. What is meant by a code walkthrough? What are some of the important types of errors checked during code walkthrough? Give one example of each of these types of errors.
26. Answer the following. Show the steps of your computation, and justify your answer in each case.
    (a) Suppose a program contains N decision points, each of which has two branches.
    How many test cases are necessary for branch testing?
    (b) If there are M choices at each decision point, how many test cases are needed for branch testing? Is it possible to achieve branch coverage using a smaller number of test cases than you have answered depending on the branch conditions?
    (c) A program consists of m sequence type of statements, n decision statements, and p iterative statements. Determine the number of test cases required to achieve decision coverage and path coverage respectively.
    (d) For a program containing N binary branches how many test cases are necessary for path coverage?
    (e) For a program containing N number of M-ary branches, how many test cases are necessary for path coverage?
27. Suppose two programmers are assigned the same programming problem and they develop this independently. Explain how can you compare their programs with respect to:
    (a) Path testing effort,
    (b) Understanding difficulty

(c) Number of latent bugs, and
(d) Reliability.

28. Usually large software products are tested at three different testing levels, i.e.,, unit testing, integration testing, and system testing. What would be the disadvantage of performing a thorough testing only after the system has been completely developed, e.g., detect all the defects of the product during system testing?
29. What do you understand by system testing? What are the different kinds of system testing that are usually performed on large software products?
30. Is system testing of object-oriented programs any different from that for the procedural programs? Explain your answer.
31. Is integration testing of object-oriented programs any different from that for the procedural programs? Explain your answer.
32. Using suitable examples, explain how test cases can be designed for an object-oriented program from its class diagram.
33. Using suitable examples, explain how test cases can be designed for an object-oriented program from its sequence diagrams.
34. Distinguish between alpha, beta, and acceptance testing. How are the test cases designed for these tests? Are the test cases for the three types of tests necessarily identical? Explain your answer.
35. Usability of a software product is tested during which type of testing: unit, integration, or system testing? How is usability tested?
36. Suppose a developed software has successfully passed all the three levels of testing, i.e., unit testing, integration testing, and system testing. Can we claim that the software is defect free? Justify your answer.
37. Distinguish among a test case, a test suite, a test scenario, and a test script.
38. Distinguish between the static and dynamic analysis of a program. Explain at least one metric that a static analysis tool reports and at least one metric that a dynamic analysis tool reports. How are these metrics useful?
39. What are the important results that are usually reported by a static analysis tool and dynamic analysis tool when applied to a program under development? How are these results useful?

40. What do you understand by automatic program analysis? Give a broad classification of the different types of program analysis tools used during program development. What are the different types of information produced by each type of tool?
41. Design the black-box test suite for a function that checks whether a character string (of up to twenty-five characters in length) is a palindrome.
42. Design the black-box test suite for a function that takes the name of a book as input and searches a file containing the names of the books available in the Library and displays the details of the book if the book is available in the library otherwise displays the message "book not available".
43. Why is it important to properly document a software? What are the different ways of documenting a software product?
44. What do you understand by the clean room strategy? What are its advantages?
45. How can you compute the cyclomatic complexity of a program? How is cyclomatic complexity useful in program testing?
46. Suppose in order to estimate the number of latent errors in a program, you seed it with hudred errors of different kinds. After testing the software using its full test set, you discover only eighty of the introduced errors. You discover fifteen other errors also. Estimate the number of latent errors in the software. What are the limitations of the error seeding method?
47. What is stress testing? Why is stress testing applicable to only certain types of systems?
48. What do you understand by unit testing? Write the code for a module that contains the functions to implement the functionality of a bounded stack of hundred integers. The queue elements are loaded from and stored into an Oracle database system—Assume that the stack support the operations: push, pop, and is-empty. Design the unit test cases for the module.
49. What do you understand by the term integration testing? Which types of defects are uncovered during integration testing? What are the different types of integration testing methods that can be used to carry out integration testing of a large software product? Compare the merits and demerits of these different integration testing strategies.

50. Discuss how you would perform system testing of a software that implements a bounded queue of positive integral elements. Assume that the queue supports only the functions insert an element, delete an element, and find an element.
51. What do you understand by side effects of a function call? Give one example of a side effect. Why are obscure side effects undesirable?
52. What do you mean by regression testing? When is regression testing carried out? Why is regression testing necessary? How are regression test cases designed? How is regression testing performed?
53. Do you agree with the following statement—"System testing can be considered a pure black-box test." Justify your answer.
54. What do you understand by big-bang integration testing? How is big-bang integration testing performed? What are the advantages and disadvantages of the big-bang integration testing strategy? Describe at least one situation where big-bang integration testing is desirable.
55. What is the relationship between cyclomatic complexity and program comprehensibility?
    Can you justify why such an apparent relationship exists?
56. Consider the following C function named bin-search:

```
/* num is the number the function searches in a presorted integer array
  arr */
    int bin_search(int num){
    int min,max;

    min =0;
    max =100;
    while(min!=max){
      if(arr[(min+max)/2]>num)
      max=(min+max)/2;
      else if(arr[(min+max)/2]<num)
      min=(min+max)/2;
      else return((min+max)/2);
    }
    return(-1);
  }
```

    Design a test suite for the function bin-search that satisfies the following white-box testing strategies (Show the intermediate steps in deriving the test cases):
    Statement coverage
    Branch coverage
    Condition coverage
    Path coverage

57. Consider the following C function named sort.

```
/* sort takes an integer array and sorts it in ascending order */

void sort(int a[], int n){
    int i,j;

    for(i=0;i<n-1;i++)
     for(j=i+1;j<n;j++)
       if(a[i]>a[j])
       {
         temp=a[i];
         a[i]=a[j];
         a[j]=temp;
       }

}
```

(a) Determine the cyclomatic complexity of the sort function.

(b) Design a test suite for the function sort that satisfies the following white-box testing strategies (Show the important steps in your test suite design method).

i. Statement coverage

ii. Branch coverage

iii. Condition coverage

iv. Path coverage

58. Draw the control flow graph for the following function named find-maximum. From the control flow graph, determine its cyclomatic complexity.

```
int find-maximum(int i,int j, int k){
  int max;

  if(i>j) then
   if(i>k) then max=i;
       else max=k;
   else if(j>k) max=j
    else max=k;
  return(max);
}
```

59. Suppose a C program has 240 sequence type of statements, 50

selection type of statements and 40 iteration type of statements, determine the minimum number of test cases required for path testing.

60. Compute the Fog index of this question. What does the Fog index signify? How is the Fog index useful in producing good software documentation?
61. Identify the types of defects that you would be able to detect during the following:
    (a) Code inspection
    (b) Code walkthrough
62. Design the black-box test suite for a function named **quadratic-solver**. The quadratic-solver function accepts three floating point numbers (a, b, c) representing a quadratic equation of the form $ax^2 + bx + c = 0$. It computes and displays the solution.
63. Design the black-box test suite for a function that accepts four pairs of floating point numbers representing four co-ordinate points. These four co-ordinate points represent the centres of two circles and a point on the circumference of each of the two circles. The function prints whether the two circles are intersecting, one is contained within the other, or are disjoint.
64. Design black-box test suites for a function called **find-intersection**. The function **find- intersection** takes four real numbers m_1, c_1, m_2, c_2 as its arguments representing two straight lines $y = m_1x + c_1$ and $y = m_2x + c_2$. It determines the points of intersection of the two lines. Depending on the input values to the function, it displays any one of the following messages:
    • single point of intersection
    • overlapping lines—infinite points of intersection
    • parallel lines—no points of intersection
    • invalid input values
65. Design black-box test suite for the following program. The program accepts two pairs of co-ordinates (x1,y1),(x2,y2), (x3,y3), (x4,y4). The first two points (x1,y1) and (x2,y2) represent the lower left and the upper right points of the first rectangle. The second two points (x3,y3) and (x4,y4) represent the lower left and the upper right points of the second rectangle. It is assumed that the length and width of the rectangle are parallel to either the x-axis or y-axis. The program computes the points of intersection of the two rectangles and prints their

points of intersection.

66. Design black-box test suite for a program that accepts up to ten simultaneous linear equations in up to ten independent variables and displays the solution.
67. Design the black-box test suite for the following Library Automation Software. The Library Automation Software accepts a string representing the name of a book. It checks the library catalog, and displays whether the book is listed in the catalog or not. If the book is listed in the catalog, it displays the number of copies that are currently available in the racks and the copies issued out.
68. Design the black-box test suite for a program that accepts two strings and checks if the first string is a substring of the second string and displays the number of times the first string occurs in the second string. Assume that each of the two strings has size less than twenty characters.
69. Design black-box test suite for a program that accepts a pair of points defining a straight line and another point and a float number defining the center of a circle and its radius. The program is intended to compute their points of intersection and prints them.
70. What do you understand by an executable specification language? How is it different from a traditional procedural programming language? Name an executable specification language.
71. Among the different development phases of life cycle, testing typically requires the largest manpower. Identify the main reasons behind the large manpower requirement for the testing phase.
72. What do you understand by performance testing? What are the different types of performance testing that should be performed for each of the problems outlined in the questions ten–twenty of Chapter 6?
73. Identify the types of information that should be presented in the test summary report.
74. What is the difference between top-down and bottom-up integration testing approaches?
    What are their advantages and disadvantages? Explain your answer using an example.
    Why is the mixed integration testing approach preferred by many testers?
75. What do you understand by "code review effectiveness"? How can

review effectiveness for an organisation measured quantitatively?

76. What do you understand by cyclomatic complexity of a program? How can it be measured? What are its applications in program development?

77. (a) What do you understand by static and dynamic analysis of programs? How are static and dynamic program analysis results useful?
    (b) What are the different program characteristics reported by a (i) static analysis tool
    (ii) dynamic analysis tool?
    (c) Write an algorithm for a dynamic program analyser that would compute and report the percentage of linearly independent paths covered by the test suite. Explain your algorithm. What is the computational complexity of your algorithm?

78. What are the different approaches to integration testing? Which approach is the most preferred for large software systems? Why?

79. What are the different types of errors that integration testing target to detect? Give two examples of such errors.

80. What is the difference between phased and incremental integration testing? Compare the advantages and disadvantages of these two approaches to integration testing.

81. Explain the difference between testing in the large and testing in the small. What is the purpose of each?

82. Explain the key respects in which testing of procedural and object-oriented programs differ. Do various object-oriented features make it easier to test object-oriented programs? Substantiate your answer with suitable examples.

83. Can an object-oriented program be tested by testing each of the methods, then integrating the methods, and finally performing system testing? If your answer is "yes", explain how the individual methods can be tested. If your answer is "no", explain why not?

84. What are the implications of the inheritance, polymorphism, and encapsulation features of an object-oriented program in satisfactory testing of the program?

85. What do you understand by grey-box testing? Why is grey-box testing considered important for testing object-oriented programs?

86. What are the different levels of testing object-oriented programs? What is a suitable unit for testing object-oriented programs?

87. State the Weyukar's anticomposition axiom. Give an intuitive

justification for the same.

88. How is integration testing of object-oriented programs carried out? Explain the different integration testing strategies for object-oriented programs.

89. What are alpha, beta, and acceptance testing? What are the differences among these different types of testing a software product? Explain your answer with respect to who carries out the test, when is the test carried out, and the objective of the test.

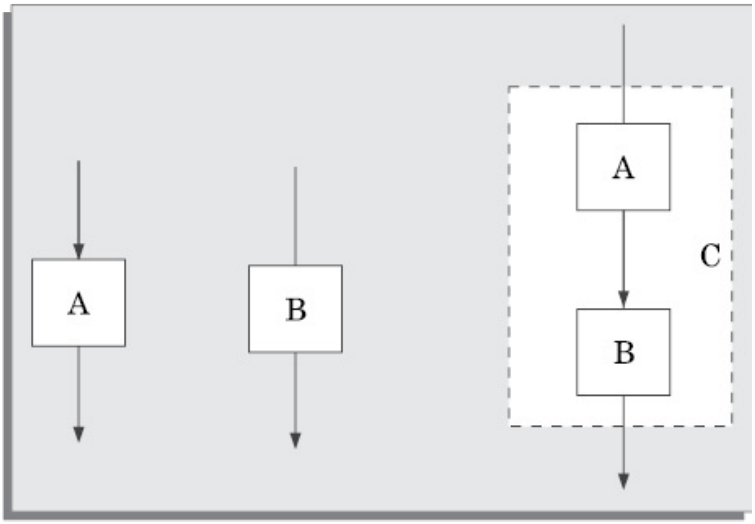

Figure 10.8: Code segment C is obtained by juxtaposing the code segments A and B.

90. What do you understand by performance testing of a software product? When is it performed? What is the objective of performance testing? What are the different types of performance testing?

91. When is the non-functional requirements tested in the life-cycle of a software product?

    How are the different non-functional requirements tested? Explain your answer with respect to various categories of non-functional requirements.

92. What do you understand by test coverage analysis? What are the uses of test coverage analysis? Define at least two test coverage metrics.

93. What do you understand by a symbolic debugger? How is debugging performed by a symbolic debugger? What are the other popular techniques for debugging?

94. What do you understand by data flow testing? How is data flow testing performed? Is it possible to design data flow test cases manually? Explain your answer.

95. What is the difference between black-box and white-box testing? During unit testing, can black-box testing be skipped, if one is planning to perform a thorough white-box testing? Justify your answer.
96. Distinguish between the static and dynamic analysis of a program. Explain at least one metric that a static analysis tool reports and at least one metric that a dynamic analysis tool reports. How are these metrics useful?
97. Suppose the cyclomatic complexities of code segments A and B (shown in Figure 10.8) are m and n respectively. What would be the cyclomatic complexity of the code segment C which has been obtained by juxtaposing the code segments A and B?

1 Manpower turnover is the software industry jargon for denoting the unusually high rate at which personnel attrition occurs (i.e., personnel leave an organisation).

Chapter

11

SOFTWARE RELIABILITY AND QUALITY MANAGEMENT

Reliability of a software product is an important concern for most users. Users not only want the products they purchase to be highly reliable, but for certain categories of products they may even require a quantitative guarantee on the reliability of the product before making their buying decision. This may especially be true for safety-critical and embedded software products. However, as we discuss in this Chapter, it is very difficult to accurately measure the reliability of any software product. One of the main problems encountered while quantitatively measuring the reliability of a software product is the fact that reliability is observer-dependent. That is, different groups of users may arrive at different reliability estimates for the same product. Besides this, several other problems (such as frequently changing reliability values due to bug corrections) make accurate measurement of the reliability of a software product difficult. We investigate these issues in this chapter. Even though no entirely satisfactory metric to measure the reliability of a software product exists, we shall discuss some metrics that are being used at present to quantify the reliability of a software product. We shall also address the problem of reliability growth modelling and examine how to predict when (and if at all) a given level of reliability will be achieved. We shall also examine the statistical testing approach to reliability estimation.

In this chapter, in addition to software reliability issues, we shall also discuss various issues associated with software quality assurance (SQA). Software quality assurance (SQA) has emerged as one of the most talked about topics in recent years in software industry circle. The major aim of SQA is to help an organisation develop high quality software products in a

repeatable manner. A software development organisation can be called *repeatable* when its software development process is person-independent. That is, the success of a project does not depend on who exactly are the team members of the project. Besides, the quality of the developed software and the cost of development are important issues addressed by SQA. In this chapter, we first discuss a few important issues concerning software reliability measurement and prediction before starting our discussion on software quality assurance.

11.1 SOFTWARE RELIABILITY

The reliability of a software product essentially denotes its *trustworthiness or dependability*. Alternatively, the reliability of a software product can also be defined as the probability of the product working "correctly" over a given period of time.

Intuitively, it is obvious that a software product having a large number of defects is unreliable. It is also very reasonable to assume that the reliability of a system improves, as the number of defects in it is reduced. It would have been very nice if we could mathematically characterise this relationship between reliability and the number of bugs present in the system using a simple closed form expression. Unfortunately, it is very difficult to characterise the observed reliability of a system in terms of the number of latent defects in the system using a simple mathematical expression. To get an insight into this issue, consider the following. Removing errors from those parts of a software product that are very infrequently executed, makes little difference to the perceived reliability of the product. It has been experimentally observed by analysing the behaviour of a large number of programs that 90 per cent of the execution time of a typical program is spent in executing only 10 per cent of the instructions in the program. The *most used* 10 per cent instructions are often called the *core*[1] of a program. The rest 90 per cent of the program statements are called *non-core* and are on the average executed only for 10 per cent of the total execution time. It therefore may not be very surprising to note that removing 60 per cent product defects from the least used parts of a system would typically result in only 3 per cent improvement to the product reliability. It is clear that the quantity by which the overall reliability of a program improves due to the correction of a single error depends on how frequently the instruction having the error is executed. If an error is removed from an instruction that is frequently executed (i.e.,

belonging to the core of the program), then this would show up as a large improvement to the reliability figure. On the other hand, removing errors from parts of the program that are rarely used, may not cause any appreciable change to the reliability of the product.

Based on the above discussion we can say that reliability of a product depends not only on the number of latent errors but also on the the exact location of the errors. Apart from this, reliability also depends upon how the product is used, or on its *execution profile*. If the users execute only those features of a program that are "correctly" implemented, none of the errors will be exposed and the perceived reliability of the product will be high. On the other hand, if only those functions of the software which contain errors are invoked, then a large number of failures will be observed and the perceived reliability of the system will be very low. Different categories of users of a software product typically execute different functions of a software product. For example, for a Library Automation Software the library members would use functionalities such as issue book, search book, etc., on the other hand the librarian would normally execute features such as create member, create book record, delete member record, etc. So defects which show up for the librarian, may not show up for the members. Suppose the functions of a Library Automation Software which the library members use are error-free; and functions used by the Librarian have many bugs. Then, these two categories of users would have very different opinions about the reliability of the software. Therefore,

Based on the above discussions, we can summarise the main reasons that make software reliability more difficult to measure than hardware reliability:

- The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
- The perceived reliability of a software product is observer-dependent.
- The reliability of a product keeps changing as errors are detected and fixed.

In the following subsection, we shall discuss why software reliability measurement is a harder problem than hardware reliability measurement.

11.1.1 Hardware versus Software Reliability

An important characteristic feature that sets hardware and software reliability issues apart is the difference between their failure patterns.

Hardware components fail due to very different reasons as compared to software components. Hardware components fail mostly due to wear and tear, whereas software components fail due to bugs.

A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix a hardware fault, one has to either replace or repair the failed part. In contrast, a software product would continue to fail until the error is tracked down and either the design or the code is changed to fix the bug. For this reason, when a hardware part is repaired its reliability would be maintained at the level that existed before the failure occurred; whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug fix introduces new errors). To put this fact in a different perspective, hardware reliability study is concerned with stability (for example, the inter-failure times remain constant). On the other hand, the aim of software reliability study would be reliability growth (that is, increase in inter-failure times).

A comparison of the changes in failure rate over the product life time for a typical hardware product as well as a software product are sketched in Figure 11.1. Observe that the plot of change of reliability with time for a hardware component (Figure 11.1(a)) appears like a "bath tub". For a software component the failure rate is initially high, but decreases as the faulty components identified are either repaired or replaced. The system then enters its useful life, where the rate of failure is almost constant. After some time (called product life time) the major components wear out, and the failure rate increases. The initial failures are usually covered through manufacturer's warranty. A corollary of this observation (though a digression from our topic of discussion) is that it may be unwise to buy a product (even at a good discount to its face value) towards the end of its life time, That is, one need not feel happy to buy a ten year old car at one tenth of the price of a new car, since it would be near the rising edge of the bath tub curve, and one would have to spend unduly large time, effort, and money on repairing and end up as the loser. In contrast to the hardware products, the software product show the highest failure rate just after purchase and installation (see the initial portion of the plot in Figure 11.1 (b)). As the system is used, more and more errors are identified and removed resulting in reduced failure rate. This error removal continues at a slower pace during the useful life of the product. As the software becomes obsolete no more error correction occurs and the failure rate remains unchanged.

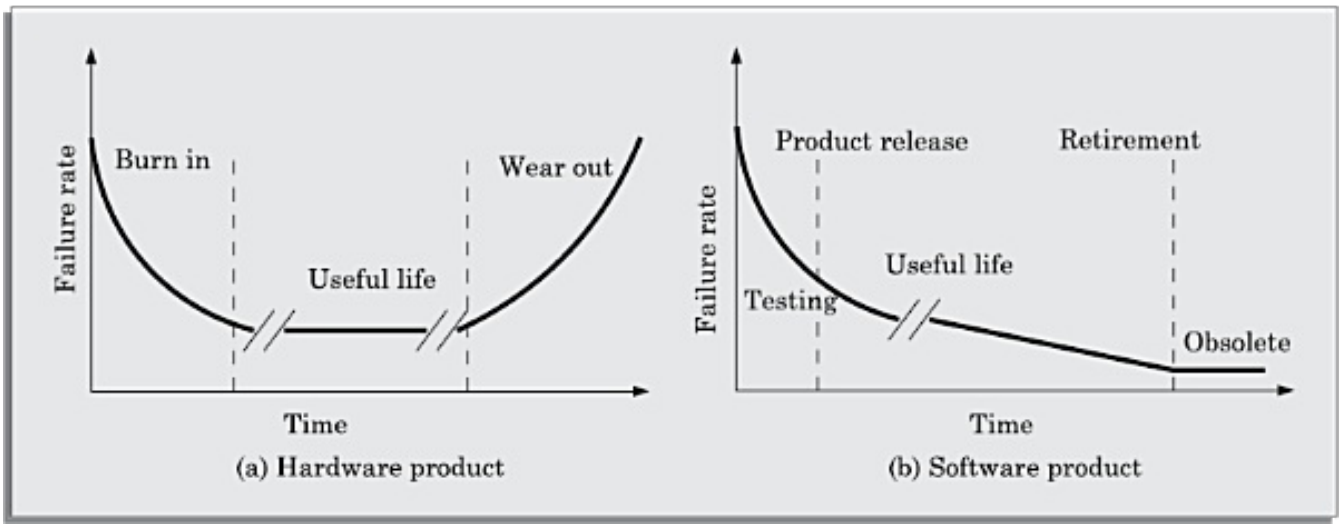

Figure 11.1: Change in failure rate of a product.

11.1.2 Reliability Metrics of Software Products

The reliability requirements for different categories of software products may be different. For this reason, it is necessary that the level of reliability required for a software product should be specified in the software requirements specification (SRS) document. In order to be able to do this, we need some metrics to quantitatively express the reliability of a software product. A good reliability measure should be observer-independent, so that different people can agree on the degree of reliability a system has. However, in practice, it is very difficult to formulate a metric using which precise reliability measurement would be possible. In the absence of such measures, we discuss six metrics that correlate with reliability as follows:

Rate of occurrence of failure (ROCOF): ROCOF measures the frequency of occurrence of failures. ROCOF measure of a software product can be obtained by observing the behaviour of a software product in operation over a specified time interval and then calculating the ROCOF value as the ratio of the total number of failures observed and the duration of observation. However, many software products do not run continuously (unlike a car or a mixer), but deliver certain service when a demand is placed on them. For example, a library software is idle until a book issue request is made. Therefore, for a typical software product such as a pay-roll software, applicability of ROCOF is very limited.

Mean time to failure (MTTF): MTTF is the time between two successive failures, averaged over a large number of failures. To measure MTTF, we can record the failure data for n failures. Let the failures occur at the time

instants t_1, t_2, ..., t_n. Then, MTTF can be calculated as

$$\sum_{i=1}^{n} \frac{t_{i+1}-t_i}{(n-1)}.$$

It is important to note that only run time is considered in the time measurements. That is, the time for which the system is down to fix the error, the boot time, etc. are not taken into account in the time measurements and the clock is stopped at these times.

Mean time to repair (MTTR): Once failure occurs, some time is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.

Mean time between failure (MTBF): The MTTF and MTTR metrics can be combined to get the MTBF metric: MTBF=MTTF+MTTR. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours. In this case, the time measurements are real time and not the execution time as in MTTF

Probability of failure on demand (POFOD): Unlike the other metrics discussed, this metric does not explicitly involve time measurements. POFOD measures the likelihood of the system failing when a service request is made. For example, a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in a failure. We have already mentioned that the reliability of a software product should be determined through specific service invocations, rather than making the software run continuously. Thus, POFOD metric is very appropriate for software products that are not required to run continuously.

Availability: Availability of a system is a measure of how likely would the system be available for use over a given period of time. This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (down time) of a system when a failure occurs. This metric is important for systems such as telecommunication systems, and operating systems, and embedded controllers, etc. which are supposed to be never down and where repair and restart time are significant and loss of service during that time cannot be overlooked.

Shortcomings of reliability metrics of software products

All the above reliability metrics suffer from several shortcomings as far as their use in software reliability measurement is concerned. One of the reasons is that these metrics are centered around the probability of

occurrence of system failures but take no account of the consequences of failures. That is, these reliability models do not distinguish the relative severity of different failures. Failures which are transient and whose consequences are not serious are in practice of little concern in the operational use of a software product. These types of failures can at best be minor irritants. On the other hand, more severe types of failures may render the system totally unusable. In order to estimate the reliability of a software product more accurately, it is necessary to classify various types of failures. Please note that the different classes of failures may not be mutually exclusive. The classification is based on widely different set of criteria. As a result, a failure type can at the same time belong to more than one class. A scheme of classification of failures is as follows:

Transient: Transient failures occur only for certain input values while invoking a function of the system.

Permanent: Permanent failures occur for all input values while invoking a function of the system.

Recoverable: When a recoverable failure occurs, the system can recover without having to shutdown and restart the system (with or without operator intervention).

Unrecoverable: In unrecoverable failures, the system may need to be restarted.

Cosmetic: These classes of failures cause only minor irritations, and do not lead to incorrect results. An example of a cosmetic failure is the situation where the mouse button has to be clicked twice instead of once to invoke a given function through the graphical user interface.

11.1.3 Reliability Growth Modelling

A reliability growth model is a mathematical model of how software reliability improves as errors are detected and repaired.

> A reliability growth model can be used to predict when (or if at all) a particular level of reliability is likely to be attained. Thus, reliability growth modelling can be used to determine when to stop testing to attain a given reliability level.

Although several different reliability growth models have been proposed, in this text we will discuss only two very simple reliability growth models.

Jelinski and Moranda model

The simplest reliability growth model is a step function model where it is assumed that the reliability increases by a constant increment each time an error is detected and repaired. Such a model is shown in Figure 11.2. However, this simple model of reliability which implicitly assumes that all errors contribute equally to reliability growth, is highly unrealistic since we already know that correction of different errors contribute differently to reliability growth.

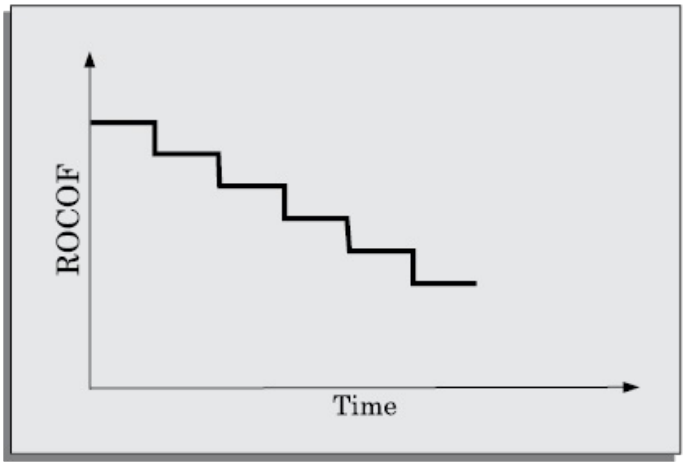

Figure 11.2: Step function model of reliability growth.

Littlewood and Verall's model

This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors. It also models the fact that as errors are repaired, the average improvement to the product reliability per repair decreases. It treats an error's contribution to reliability improvement to be an independent random variable having Gamma distribution. This distribution models the fact that error corrections with large contributions to reliability growth are removed first. This represents diminishing return as test continues.

There are more complex reliability growth models, which give more accurate approximations to the reliability growth. However, these models are out of scope of this text.

11.2 STATISTICAL TESTING

Statistical testing is a testing process whose objective is to determine the reliability of the product rather than discovering errors. The test cases designed for statistical testing with an entirely different objective

from those of conventional testing. To carry out statistical testing, we need to first define the operation profile of the product.

Operation profile: Different categories of users may use a software product for very different purposes. For example, a librarian might use the Library Automation Software to create member records, delete member records, add books to the library, etc., whereas a library member might use software to query about the availability of a book, and to issue and return books. Formally, we can define the operation profile of a software as the probability of a user selecting the different functionalities of the software. If we denote the set of various functionalities offered by the software by $\{f_i\}$, the operational profile would associate with each function $\{f_i\}$ with the probability with which an average user would select $\{f_i\}$ as his next function to use. Thus, we can think of the operation profile as assigning a probability value pi to each functionality fi of the software.

How to define the operation profile for a product?

We need to divide the input data into a number of input classes. For example, for a graphical editor software, we might divide the input into data associated with the edit, print, and file operations. We then need to assign a probability value to each input class; to signify the probability for an input value from that class to be selected. The operation profile of a software product can be determined by observing and analysing the usage pattern of the software by a number of users.

11.2.1 Steps in Statistical Testing

The first step is to determine the operation profile of the software. The next step is to generate a set of test data corresponding to the determined operation profile. The third step is to apply the test cases to the software and record the time between each failure. After a statistically significant number of failures have been observed, the reliability can be computed.

For accurate results, statistical testing requires some fundamental assumptions to be satisfied. It requires a statistically significant number of test cases to be used. It further requires that a small percentage of test inputs that are likely to cause system failure to be included. Now let us discuss the implications of these assumptions.

It is straight forward to generate test cases for the common types of inputs, since one can easily write a test case generator program which can

automatically generate these test cases. However, it is also required that a statistically significant percentage of the unlikely inputs should also be included in the test suite. Creating these unlikely inputs using a test case generator is very difficult.

Pros and cons of statistical testing

Statistical testing allows one to concentrate on testing parts of the system that are most likely to be used. Therefore, it results in a system that the users can find to be more reliable (than actually it is!). Also, the reliability estimation arrived by using statistical testing is more accurate compared to those of other methods discussed. However, it is not easy to perform the statistical testing satisfactorily due to the following two reasons. There is no simple and repeatable way of defining operation profiles. Also, the the number of test cases with which the system is to be tested should be statistically significant.

11.3 SOFTWARE QUALITY

Traditionally, the quality of a product is defined in terms of its fitness of purpose. That is, a good quality product does exactly what the users want it to do, since for almost every product, fitness of purpose is interpreted in terms of satisfaction of the requirements laid down in the SRS document. Although "fitness of purpose" is a satisfactory definition of quality for many products such as a car, a table fan, a grinding machine, etc.—"fitness of purpose" is not a wholly satisfactory definition of quality for software products. To give an example of why this is so, consider a software product that is functionally correct. That is, it correctly performs all the functions that have been specified in its SRS document. Even though it may be functionally correct, we cannot consider it to be a quality product, if it has an almost unusable user interface. Another example is that of a product which does everything that the users wanted but has an almost incomprehensible and unmaintainable code. Therefore, the traditional concept of quality as "fitness of purpose" for software products is not wholly satisfactory.

Unlike hardware products, software lasts a long time, in the sense that it keeps evolving to accommodate changed circumstances. The modern view of a quality associates with a software product several quality factors (or attributes) such as the following:

Portability: A software product is said to be portable, if it can be easily

made to work in different hardware and operating system environments, and easily interface with external hardware devices and software products.

Usability: A software product has good usability, if different categories of users (i.e., both expert and novice users) can easily invoke the functions of the product.

Reusability: A software product has good reusability, if different modules of the product can easily be reused to develop new products.

Correctness: A software product is correct, if different requirements as specified in the SRS document have been correctly implemented.

Maintainability: A software product is maintainable, if errors can be easily corrected as and when they show up, new functions can be easily added to the product, and the functionalities of the product can be easily modified, etc.

McCall's quality factors

McCall distinguishes two levels of quality attributes [McCall]. The higher-level attributes, known as quality factor s or external attributes can only be measured indirectly. The second-level quality attributes are called quality criteria. Quality criteria can be measured directly, either objectively or subjectively. By combining the ratings of several criteria, we can either obtain a rating for the quality factors, or the extent to which they are satisfied. For example, the reliability cannot be measured directly, but by measuring the number of defects encountered over a period of time. Thus, reliability is a higher-level quality factor and number of defects is a low-level quality factor.

ISO 9126

ISO 9126 defines a set of hierarchical quality characteristics. Each subcharacteristic in this is related to exactly one quality characteristic. This is in contrast to the McCall's quality attributes that are heavily interrelated. Another difference is that the ISO characteristic strictly refers to a software product, whereas McCall's attributes capture process quality issues as well.

The users as well as the managers tend to be interested in the higher-level quality attributes (quality factors).

11.4 SOFTWARE QUALITY MANAGEMENT SYSTEM

A quality management system (often referred to as quality system) is

the principal methodology used by organisations to ensure that the products they develop have the desired quality. In the following subsections, we briefly discuss some of the important issues associated with a quality system:

Managerial structure and individual responsibilities

A quality system is the responsibility of the organisation as a whole. However, every organisation has a separate quality department to perform several quality system activities. The quality system of an organisation should have the full support of the top management. Without support for the quality system at a high level in a company, few members of staff will take the quality system seriously.

Quality system activities

The quality system activities encompass the following:

- Auditing of projects to check if the processes are being followed.
- Collect process and product metrics and analyse them to check if quality goals are being met.
- Review of the quality system to make it more effective.
- Development of standards, procedures, and guidelines.
- Produce reports for the top management summarising the effectiveness of the quality system in the organisation.

A good quality system must be well documented. Without a properly documented quality system, the application of quality controls and procedures become ad hoc, resulting in large variations in the quality of the products delivered. Also, an undocumented quality system sends clear messages to the staff about the attitude of the organisation towards quality assurance. International standards such as ISO 9000 provide guidance on how to organise a quality system.

11.4.1 Evolution of Quality Systems

Quality systems have rapidly evolved over the last six decades. Prior to World War II, the usual method to produce quality products was to inspect the finished products to eliminate defective products. For example, a company manufacturing nuts and bolts would inspect its finished goods and would reject those nuts and bolts that are outside certain specified tolerance range.

Since that time, quality systems of organisations have undergone four stages of evolution as shown in Figure 11.3. The initial product inspection method gave way to quality control (QC) principles.

> Quality control (QC) focuses not only on detecting the defective products and eliminating them, but also on determining the causes behind the defects, so that the product rejection rate can be reduced.

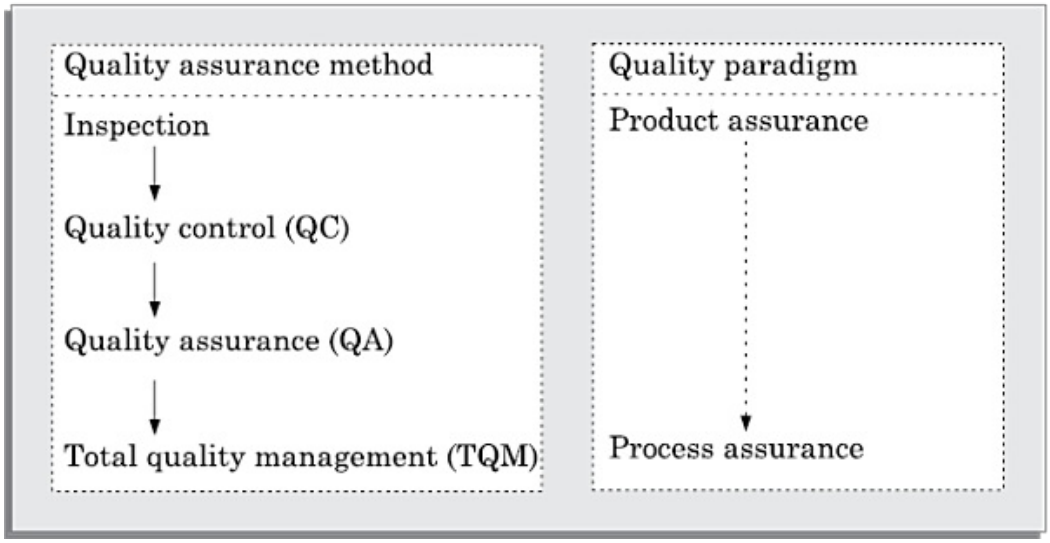

Figure 11.3: Evolution of quality system and corresponding shift in the quality paradigm.

Thus, quality control aims at correcting the causes of errors and not just rejecting the defective products. The next breakthrough in quality systems, was the development of the quality assurance (QA) principles.

> The basic premise of modern quality assurance is that if an organisation's processes are good and are followed rigorously, then the products are bound to be of good quality.

The modern quality assurance paradigm includes guidance for recognising, defining, analysing, and improving the production process. Total quality management (TQM) advocates that the process followed by an organisation must continuously be improved through process measurements. TQM goes a step further than quality assurance and aims at continuous process improvement. TQM goes beyond documenting processes to optimising them through redesign. A term related to TQM is business process re-engineering (BPR), which is aims at re-engineering the way business is carried out in an organisation, whereas our focus in this text is re-engineering of the software development process. From the above discussion, we can say that over the last six decades or so, the quality paradigm has shifted from product assurance to process assurance (see Figure 11.3).

11.4.2 Product Metrics versus Process Metrics

All modern quality systems lay emphasis on collection of certain product and process metrics during product development. Let us first understand the basic differences between product and process metrics.

> Product metrics help measure the characteristics of a product being developed, whereas process metrics help measure how a process is performing.

Examples of product metrics are LOC and function point to measure size, PM (person- month) to measure the effort required to develop it, months to measure the time required to develop the product, time complexity of the algorithms, etc. Examples of process metrics are review effectiveness, average number of defects found per hour of inspection, average defect correction time, productivity, average number of failures detected during testing per LOC, number of latent defects per line of code in the developed product.

11.5 ISO 9000

International standards organisation (ISO) is a consortium of 63 countries established to formulate and foster standardisation. ISO published its 9000 series of standards in 1987.

11.5.1 What is ISO 9000 Certification?

ISO 9000 certification serves as a reference for contract between independent parties. In particular, a company awarding a development contract can form his opinion about the possible vendor performance based on whether the vendor has obtained ISO 9000 certification or not. In this context, the ISO 9000 standard specifies the guidelines for maintaining a quality system. We have already seen that the quality system of an organisation applies to all its activities related to its products or services. The ISO standard addresses both operational aspects (that is, the process) and organisational aspects such as responsibilities, reporting, etc. In a nutshell, ISO 9000 specifies a set of recommendations for repeatable and high quality product development. It is important to realise that ISO 9000 standard is a set of guidelines for the production process and is not directly concerned about the product it self.

ISO 9000 is a series of three standards—ISO 9001, ISO 9002, and ISO

9003.

> The ISO 9000 series of standards are based on the premise that if a proper process is followed for production, then good quality products are bound to follow automatically.

The types of software companies to which the different ISO standards apply are as follows:

ISO 9001: This standard applies to the organisations engaged in design, development, production, and servicing of goods. This is the standard that is applicable to most software development organisations.

ISO 9002: This standard applies to those organisations which do not design products but are only involved in production. Examples of this category of industries include steel and car manufacturing industries who buy the product and plant designs from external sources and are involved in only manufacturing those products. Therefore, ISO 9002 is not applicable to software development organisations.

ISO 9003: This standard applies to organisations involved only in installation and testing of products.

11.5.2 ISO 9000 for Software Industry

ISO 9000 is a generic standard that is applicable to a large gamut of industries, starting from a steel manufacturing industry to a service rendering company. Therefore, many of the clauses of the ISO 9000 documents are written using generic terminologies and it is very difficult to interpret them in the context of software development organisations. An important reason behind such a situation is the fact that software development is in many respects radically different from the development of other types of products. Two major differences between software development and development of other kinds of products are as follows:

- Software is intangible and therefore difficult to control. It means that software would not be visible to the user until the development is complete and the software is up and running. It is difficult to control and manage anything that you cannot see and feel. In contrast, in any other type of product manufacturing such as car manufacturing, you can see a product being developed through various stages such as fitting engine, fitting doors, etc. Therefore, it becomes easy to

accurately determine how much work has been completed and to estimate how much more time will it take.

- During software development, the only raw material consumed is data. In contrast, large quantities of raw materials are consumed during the development of any other product. As an example, consider a steel making company. The company would consume large amounts of raw material such as iron-ore, coal, lime, manganese, etc. Not surprisingly then, many clauses of ISO 9000 standards are concerned with raw material control. These clauses are obviously not relevant for software development organisations.

Due to such radical differences between software and other types of product development, it was difficult to interpret various clauses of the original ISO standard in the context of software industry. Therefore, ISO released a separate document called ISO 9000 part-3 in 1991 to help interpret the ISO standard for software industry. At present, official guidance is inadequate regarding the interpretation of various clauses of ISO 9000 standard in the context of software industry and one has to keep on cross referencing the ISO 9000-3 document.

11.5.3 Why Get ISO 9000 Certification?

There is a mad scramble among software development organisations for obtaining ISO certification due to the benefits it offers. Let us examine some of the benefits that accrue to organisations obtaining ISO certification:

- Confidence of customers in an organisation increases when the organisation qualifies for ISO 9001 certification. This is especially true in the international market. In fact, many organisations awarding international software development contracts insist that the development organisation have ISO 9000 certification. For this reason, it is vital for software organisations involved in software export to obtain ISO 9000 certification.
- ISO 9000 requires a well-documented software production process to be in place. A well- documented software production process contributes to repeatable and higher quality of the developed software.
- ISO 9000 makes the development process focused, efficient, and cost-effective.

- ISO 9000 certification points out the weak points of an organisations and recommends remedial action.
- ISO 9000 sets the basic framework for the development of an optimal process and TQM.

11.5.4 How to Get ISO 9000 Certification?

An organisation intending to obtain ISO 9000 certification applies to a ISO 9000 registrar for registration. The ISO 9000 registration process consists of the following stages:

Application stage: Once an organisation decides to go for ISO 9000 certification, it applies to a registrar for registration.

Pre-assessment: During this stage the registrar makes a rough assessment of the organisation.

Document review and adequacy audit: During this stage, the registrar reviews the documents submitted by the organisation and makes suggestions for possible improvements.

Compliance audit: During this stage, the registrar checks whether the suggestions made by it during review have been complied to by the organisation or not.

Registration: The registrar awards the ISO 9000 certificate after successful completion of all previous phases.

Continued surveillance: The registrar continues monitoring the organisation periodically.

ISO mandates that a certified organisation can use the certificate for corporate advertisements but cannot use the certificate for advertising any of its products.

This is probably due to the fact that the ISO 9000 certificate is issued for an organisation's process and not to any specific product of the organisation. An organisation using ISO certificate for product advertisements faces the risk of withdrawal of the certificate. In India, ISO 9000 certification is offered by BIS (Bureau of Indian Standards), STQC (Standardisation, testing, and quality control), and IRQS (Indian Register Quality System). IRQS has been accredited by the Dutch council of certifying bodies (RVC).

11.5.5 Summary of ISO 9001 Requirements

A summary of the main requirements of ISO 9001 as they relate of

software development are as follows:
Section numbers in brackets correspond to those in the standard itself:

Management responsibility (4.1)

- The management must have an effective quality policy.
- The responsibility and authority of all those whose work affects quality must be defined and documented.
- A management representative, independent of the development process, must be responsible for the quality system. This requirement probably has been put down so that the person responsible for the quality system can work in an unbiased manner.
- The effectiveness of the quality system must be periodically reviewed by audits.

Quality system (4.2)

A quality system must be maintained and documented.

Contract reviews (4.3)

Before entering into a contract, an organisation must review the contract to ensure that it is understood, and that the organisation has the necessary capability for carrying out its obligations.

Design control (4.4)

- The design process must be properly controlled, this includes controlling coding also. This requirement means that a good configuration control system must be in place.
- Design inputs must be verified as adequate.
- Design must be verified.
- Design output must be of required quality.
- Design changes must be controlled.

Document control (4.5)

- There must be proper procedures for document approval, issue and removal.
- Document changes must be controlled. Thus, use of some configuration management tools is necessary.

Purchasing (4.6)

Purchased material, including bought-in software must be checked for conforming to requirements.

Purchaser supplied product (4.7)

Material supplied by a purchaser, for example, client-provided software must be properly managed and checked.

Product identification (4.8)

The product must be identifiable at all stages of the process. In software terms this means configuration management.

Process control (4.9)

- The development must be properly managed.
- Quality requirement must be identified in a quality plan.

Inspection and testing (4.10)

In software terms this requires effective testing i.e., unit testing, integration testing and system testing. Test records must be maintained.

Inspection, measuring and test equipment (4.11)

If integration, measuring, and test equipments are used, they must be properly maintained and calibrated.

Inspection and test status (4.12)

The status of an item must be identified. In software terms this implies configuration management and release control.

Control of non-conforming product (4.13)

In software terms, this means keeping untested or faulty software out of the released product, or other places whether it might cause damage.

Corrective action (4.14)

This requirement is both about correcting errors when found, and also investigating why the errors occurred and improving the process to prevent occurrences. If an error occurs despite the quality system, the

system needs improvement.

Handling (4.15)

This clause deals with the storage, packing, and delivery of the software product.

Quality records (4.16)

Recording the steps taken to control the quality of the process is essential in order to be able to confirm that they have actually taken place.

Quality audits (4.17)

Audits of the quality system must be carried out to ensure that it is effective.

Training (4.18)

Training needs must be identified and met.

Various ISO 9001 requirements are largely common sense. Official guidance on the

interpretation of ISO 9001 is inadequate at the present time, and taking expert advice is usually worthwhile.

11.5.6 Salient Features of ISO 9001 Requirements

In subsection 11.5.5 we pointed out the various requirements for the ISO 9001 certification. We can summarise the salient features all the the requirements as follows:

Document control: All documents concerned with the development of a software product should be properly managed, authorised, and controlled. This requires a configuration management system to be in place.

Planning: Proper plans should be prepared and then progress against these plans should be monitored.

Review: Important documents across all phases should be independently checked and reviewed for effectiveness and correctness.

Testing: The product should be tested against specification.

Organisational aspects: Several organisational aspects should be addressed e.g., management reporting of the quality team.

11.5.7 ISO 9000-2000

ISO revised the quality standards in the year 2000 to fine tune the standards. The major changes include a mechanism for continuous process improvement. There is also an increased emphasis on the role of the top management, including establishing a measurable objectives for various roles and levels of the organisation. The new standard recognises that there can be many processes in an organisation.

11.5.8 Shortcomings of ISO 9000 Certification

Even though ISO 9000 is widely being used for setting up an effective quality system in an organisation, it suffers from several shortcomings. Some of these shortcoming of the ISO 9000 certification process are the following:

- ISO 9000 requires a software production process to be adhered to, but does not guarantee the process to be of high quality. It also does not give any guideline for defining an appropriate process.
- ISO 9000 certification process is not fool-proof and no international accredition agency exists. Therefore it is likely that variations in the norms of awarding certificates can exist among the different accredition agencies and also among the registrars.
- Organisations getting ISO 9000 certification often tend to downplay domain expertise and the ingenuity of the developers. These organisations start to believe that since a good process is in place, the development results are truly person-independent. That is, any developer is as effective as any other developer in performing any particular software development activity. In manufacturing industry there is a clear link between process quality and product quality. Once a process is calibrated, it can be run again and again producing quality goods. Many areas of software development are so specialised that special expertise and experience in these areas (domain expertise) is required. Also, unlike in case of general product manufacturing, ingenuity and effectiveness of personal practices play an important part in determining the results produced by a developer. In other words, software development is a creative process and individual skills and experience are important.
- ISO 9000 does not automatically lead to continuous process improvement. In other words, it does not automatically lead to TQM.

11.6 SEI CAPABILITY MATURITY MODEL

SEI capability maturity model (SEI CMM) was proposed by Software Engineering Institute of the Carnegie Mellon University, USA. CMM is patterned after the pioneering work of Philip Crosby who published his maturity grid of five evolutionary stages in adopting quality practices in his book "Quality is Free" [Crosby79].

The Unites States Department of Defence (US DoD) is the largest buyer of software product. It often faced difficulties in vendor performances, and had to many times live with low quality products, late delivery, and cost escalations. In this context, SEI CMM was originally developed to assist the U.S. Department of Defense (DoD) in software acquisition. The rationale was to include the likely contractor performance as a factor in contract awards. Most of the major DoD contractors began CMM-based process improvement initiatives as they vied for DoD contracts. It was observed that the SEI CMM model helped organisations to improve the quality of the software they developed and therefore adoption of SEI CMM model had significant business benefits. Gradually many commercial organisations began to adopt CMM as a framework for their own internal improvement initiatives.

In simple words, CMM is a reference model for apprising the software process maturity into different levels. This can be used to predict the most likely outcome to be expected from the next project that the organisation undertakes. It must be remembered that SEI CMM can be used in two ways—capability evaluation and software process assessment. Capability evaluation and software process assessment differ in motivation, objective, and the final use of the result. Capability evaluation provides a way to assess the software process capability of an organisation. Capability evaluation is administered by the contract awarding authority, and therefore the results would indicate the likely contractor performance if the contractor is awarded a work. On the other hand, software process assessment is used by an organisation with the objective to improve its own process capability. Thus, the latter type of assessment is for purely internal use by a company.

The different levels of SEI CMM have been designed so that it is easy for an organisation to slowly build its quality system starting from scratch. SEI CMM classifies software development industries into the following five maturity levels:

Level 1: Initial

A software development organisation at this level is characterised by ad

hoc activities. Very few or no processes are defined and followed. Since software production processes are not defined, different engineers follow their own process and as a result development efforts become chaotic. Therefore, it is also called chaotic level. The success of projects depend on individual efforts and heroics. When a developer leaves the organisation, the successor would have great difficulty in understanding the process that was followed and the work completed. Also, no formal project management practices are followed. As a result, time pressure builds up towards the end of the delivery time, as a result short-cuts are tried out leading to low quality products.

Level 2: Repeatable

At this level, the basic project management practices such as tracking cost and schedule are established. Configuration management tools are used on items identified for configuration control. Size and cost estimation techniques such as function point analysis, COCOMO, etc., are used. The necessary process discipline is in place to repeat earlier success on projects with similar applications. Though there is a rough understanding among the developers about the process being followed, the process is not documented. Configuration management practices are used for all project deliverables. Please remember that opportunity to repeat a process exists only when a company produces a family of products. Since the products are very similar, the success story on development of one product can repeated for another. In a non-repeatable software development organisation, a software product development project becomes successful primarily due to the initiative, effort, brilliance, or enthusiasm displayed by certain individuals. On the other hand, in a non-repeatable software development organisation, the chances of successful completion of a software project is to a great extent depends on who the team members are. For this reason, the successful development of one product by such an organisation does not automatically imply that the next product development will be successful.

Level 3: Defined

At this level, the processes for both management and development activities are defined and documented. There is a common organisation-wide understanding of activities, roles, and responsibilities.

The processes though defined, the process and product qualities are not measured. At this level, the organisation builds up the capabilities of its employees through periodic training programs. Also, review techniques are emphasized and documented to achieve phase containment of errors. ISO 9000 aims at achieving this level.

Level 4: Managed

At this level, the focus is on software metrics. Both process and product metrics are collected. Quantitative quality goals are set for the products and at the time of completion of development it was checked whether the quantitative quality goals for the product are met. Various tools like Pareto charts, fishbone diagrams, etc. are used to measure the product and process quality. The process metrics are used to check if a project performed satisfactorily. Thus, the results of process measurements are used to evaluate project performance rather than improve the process.

Level 5: Optimising

At this stage, process and product metrics are collected. Process and product measurement data are analysed for continuous process improvement. For example, if from an analysis of the process measurement results, it is found that the code reviews are not very effective and a large number of errors are detected only during the unit testing, then the process would be fine tuned to make the review more effective. Also, the lessons learned from specific projects are incorporated into the process. Continuous process improvement is achieved both by carefully analysing the quantitative feedback from the process measurements and also from application of innovative ideas and technologies. At CMM level 5, an organisation would identify the best software engineering practices and innovations (which may be tools, methods, or processes) and would transfer these organisation-wide. Level 5 organisations usually have a department whose sole responsibility is to assimilate latest tools and technologies and propagate them organisation-wide. Since the process changes continuously, it becomes necessary to effectively manage a changing process. Therefore, level 5 organisations use configuration management techniques to manage process changes.

Except for level 1, each maturity level is characterised by several key process areas (KPAs) that indicate the areas an organisation should focus to

improve its software process to this level from the previous level. Each of the focus areas identifies a number of key practices or activities that need to be implemented. In other words, KPAs capture the focus areas of a level. The focus of each level and the corresponding key process areas are shown in the Table 11.1:

Table 11.1 Focus areas of CMM levels and Key Process Areas

CMM Level	*Focus*	*Key Process Areas (KPAs)*
Initial	Competent people	
Repeatable	Project management	Software project planning Software configuration management
Defined	Definition of processes	Process definition Training program Peer reviews
Managed	Product and process quality	Quantitative process metrics Software quality management
Optimising	Continuous process improvement	Defect prevention Process change management Technology change management

SEI CMM provides a list of key areas on which to focus to take an organisation from one level of maturity to the next. Thus, it provides a way for gradual quality improvement over several stages. Each stage has been carefully designed such that one stage enhances the capability already built up. For example, trying to implement a defined process (level 3) before a repeatable process (level 2) would be counterproductive as it becomes difficult to follow the defined process due to schedule and budget pressures.

Substantial evidence has now been accumulated which indicate that adopting SEI CMM has several business benefits. However, the organisations trying out the CMM frequently face a problem that stems from the characteristic of the CMM itself.

CMM Shortcomings: CMM does suffer from several shortcomings. The important among these are the following:

- The most frequent complaint by organisations while trying out the CMM-based process improvement initiative is that they understand what is needed to be improved, but they need more guidance about how to improve it.
- Another shortcoming (that is common to ISO 9000) is that thicker documents, more detailed information, and longer meetings are considered to be better. This is in contrast to the principles of software economics—reducing complexity and keeping the documentation to the

minimum without sacrificing the relevant details.

- Getting an accurate measure of an organisation's current maturity level is also an issue. The CMM takes an activity-based approach to measuring maturity; if you do the prescribed set of activities then you are at a certain level. There is nothing that characterises or quantifies whether you do these activities well enough to deliver the intended results.

11.6.1 Comparison Between ISO 9000 Certification and SEI/CMM

Let us compare some of the key characteristics of ISO 9000 certification and the SEI CMM model for quality appraisal:

- ISO 9000 is awarded by an international standards body. Therefore, ISO 9000 certification can be quoted by an organisation in official documents, communication with external parties, and in tender quotations. However, SEI CMM assessment is purely for internal use.
- SEI CMM was developed specifically for software industry and therefore addresses many issues which are specific to software industry alone.
- SEI CMM goes beyond quality assurance and prepares an organisation to ultimately achieve TQM. In fact, ISO 9001 aims at level 3 of SEI CMM model.
- SEI CMM model provides a list of key process areas (KPAs) on which an organisation at any maturity level needs to concentrate to take it from one maturity level to the next. Thus, it provides a way for achieving gradual quality improvement. In contrast, an organisation adopting ISO 9000 either qualifies for it or does not qualify.

11.6.2 Is SEI CMM Applicable to Small Organisations?

Highly systematic and measured approach to software development suits large organisations dealing with negotiated software, safety-critical software, etc. But, what about small organisations? These organisations typically handle applications such as small Internet, e-commerce applications, and often are without an established product range, revenue base, and experience on past projects, etc. For such organisations, a CMM-based appraisal is probably excessive. These organisations need to operate more efficiently at the lower levels of

maturity. For example, they need to practise effective project management, reviews, configuration management, etc.

11.6.3 Capability Maturity Model Integration (CMMI)

Capability maturity model integration (CMMI) is the successor of the capability maturity model (CMM). The CMM was developed from 1987 until 1997. In 2002, CMMI Version 1.1 was released. Version 1.2 followed in 2006. CMMI aimed to improve the usability of maturity models by integrating many different models into one framework.

After CMMI was first released in 1990, it was adopted and used in many domains. For example, CMMs were developed for disciplines such as systems engineering (SE-CMM), people management (PCMM), software acquisition (SA-CMM), and others. Although many organisations found these models to be useful, they also struggled with problems caused by overlap, inconsistencies, and integrating the models. In this context, CMMI is generalised to be applicable to many domains. For example, the word "software" does not appear in definitions of CMMI. This unification of various types of domains into a single model makes CMMI extremely abstract. The CMMI, like its predecessor, describes five distinct levels of maturity.

11.7 FEW OTHER IMPORTANT QUALITY STANDARDS

11.7.1 Software Process Improvement and Capability Determination (SPICE)

SPICE stands for Software Process Improvement and Capability determination. It is an ISO standard (IEC 15504). It distinguishes different kinds of processes—engineering process, management process, customer-supplier, support. For each process, it defines six capability maturity levels. It integrates existing standards to provide a single process reference model and process assessment model that addresses broad categories of enterprise processes.

11.7.2 Personal Software Process (PSP)

PSP is based on the work of David Humphrey [Hum97]. PSP is a scaled down version of industrial software process discussed in the last section. PSP is suitable for individual use. It is important to note that SEI CMM does not tell software developers how to analyse, design, code, test, or document software products, but assumes that engineers

use effective personal practices. PSP recognises that the process for individual use is different from that necessary for a team.

The quality and productivity of an engineer is to a great extent dependent on his process. PSP is a framework that helps engineers to measure and improve the way they work. It helps in developing personal skills and methods by estimating, planning, and tracking performance against plans, and provides a defined process which can be tuned by individuals.

Time measurement: PSP advocates that developers should rack the way they spend time. Because, boring activities seem longer than actual and interesting activities seem short. Therefore, the actual time spent on a task should be measured with the help of a stop-watch to get an objective picture of the time spent. For example, he may stop the clock when attending a telephone call, taking a coffee break, etc. An engineer should measure the time he spends for various development activities such as designing, writing code, testing, etc.

PSP Planning: Individuals must plan their project. Unless an individual properly plans his activities, disproportionately high effort may be spent on trivial activities and important activities may be compromised, leading to poor quality results. The developers must estimate the maximum, minimum, and the average LOC required for the product. They should use their productivity in minutes/LOC to calculate the maximum, minimum, and the average development time. They must record the plan data in a project plan summary.

The PSP is schematically shown in Figure 11.4. While carrying out the different phases, an individual must record the log data using time measurement. During post-mortem, they can compare the log data with their project plan to achieve better planning in the future projects, to improve his process, etc.

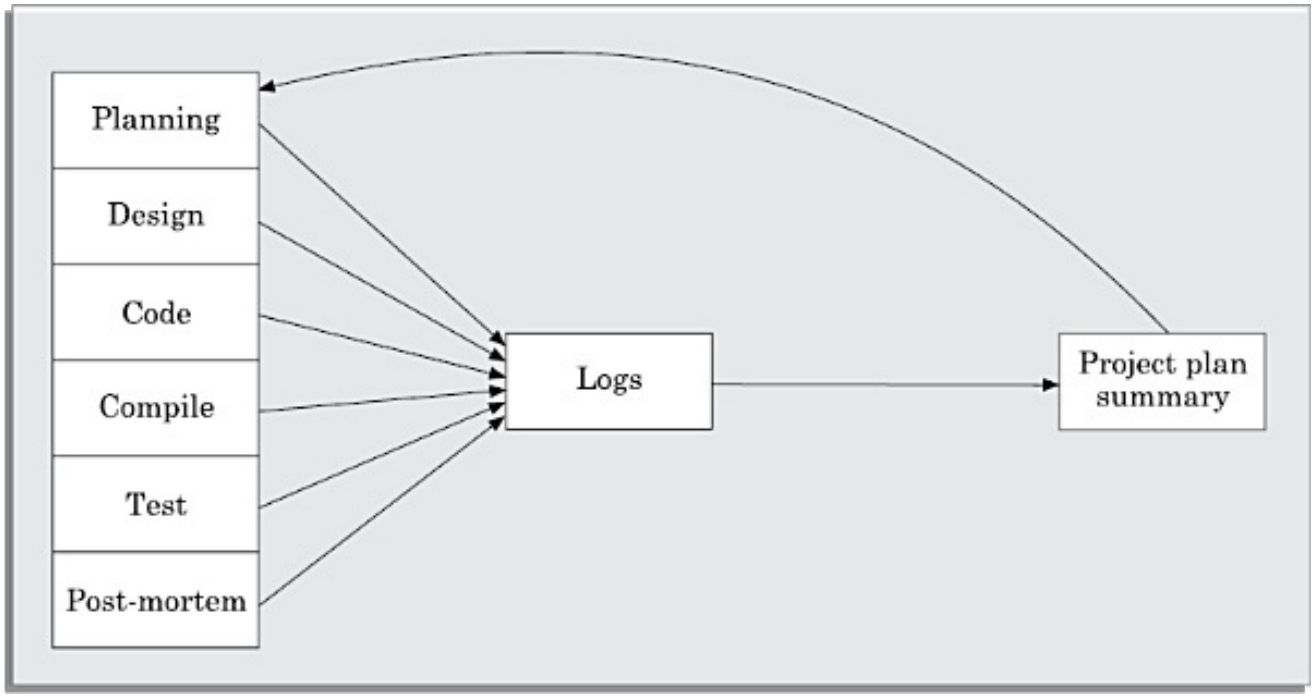

Figure 11.4: A schematic representation of PSP.

The PSP levels are summarised in Figure 11.5. PSP2 introduces defect management via the use of checklists for code and design reviews. The checklists are developed by analysing the defect data gathered from earlier projects.

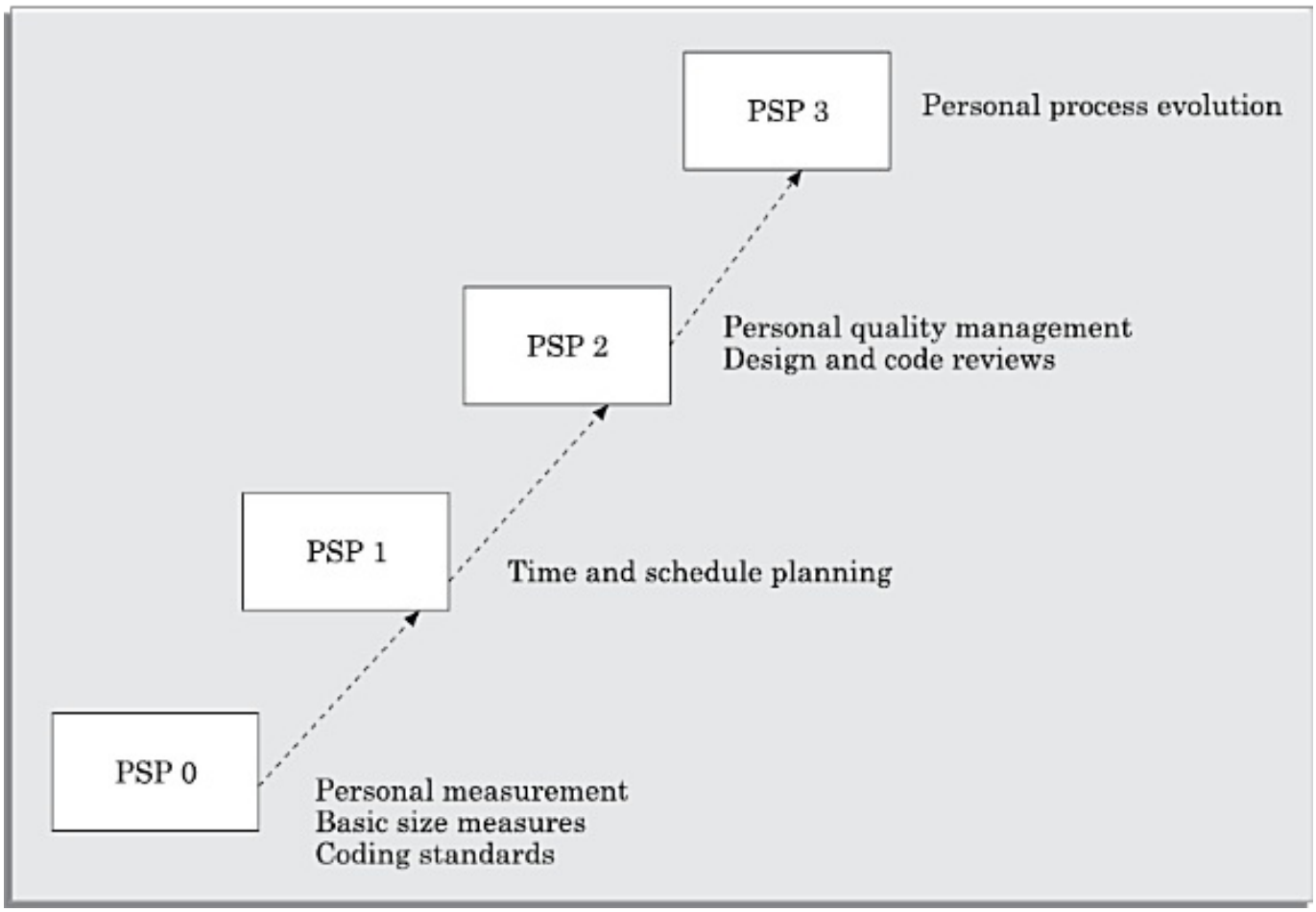

Figure 11.5: Levels of PSP.

11.8 SIX SIGMA

General Electric (GE) corporation first began Six Sigma in 1995 after Motorola and Allied Signal blazed the Six Sigma trail. Since them, thousands of companies around the world have discovered the far reaching benefits of Six Sigma. The purpose of Six Sigma is to improve processes to do things better, faster, and at lower cost. It can be used to improve every facet of business, from production, to human resources, to order entry, to technical support. Six Sigma can be used for any activity that is concerned with cost, timeliness, and quality of results. Therefore, it is applicable to virtually every industry.

Six Sigma at many organisations simply means striving for near perfection. Six Sigma is a disciplined, data-driven approach to eliminate defects in any process – from manufacturing to transactional and from product to service.

The statistical representation of Six Sigma describes quantitatively how a process is performing. To achieve Six Sigma, a process must not produce more than 3.4 defects per million opportunities. A Six Sigma defect is defined

as any system behaviour that is not as per customer specifications. Total number of Six Sigma opportunities is then the total number of chances for a defect. Process sigma can easily be calculated using a Six Sigma calculator.

The fundamental objective of the Six Sigma methodology is the implementation of a measurement-based strategy that focuses on process improvement and variation reduction through the application of Six Sigma improvement projects. This is accomplished through the use of two Six Sigma sub-methodologies—DMAIC and DMADV. The Six Sigma DMAIC process (define, measure, analyse, improve, control) is an improvement system for existing processes falling below specification and looking for incremental improvement. The Six Sigma DMADV process (define, measure, analyse, design, verify) is an improvement system used to develop new processes or products at Six Sigma quality levels. It can also be employed if a current process requires more than just incremental improvement. Both Six Sigma processes are executed by Six Sigma Green Belts and Six Sigma Black Belts, and are overseen by Six Sigma Master Black Belts.

Many frameworks exist for implementing the Six Sigma methodology. Six Sigma Consultants all over the world have also developed proprietary methodologies for implementing Six Sigma quality, based on the similar change management philosophies and applications of tools.

SUMMARY

- In this chapter, we first defined software reliability and discussed its importance.
- We pointed out that even though the number of defects remaining in a software product is correlated to its reliability, no simple relationship between the two exists. An important reason behind this is that the errors existing in the core and non-core part of a software product affect the reliability of the product differently.
- We discussed a few metrics for quantifying the reliability of a given piece of software. We have pointed out the shortcomings of these metrics and concluded that none of these metrics can be used to provide an entirely satisfactory measure of the reliability of a software product.
- We examined reliability growth modelling and its use to determine how long to test a product.
- We discussed what is meant by a quality management system and

introduced the modern concept of software quality. We also discussed the concept of total quality management (TQM).

- We pointed out that the methods of Software Process Assessment are coming more generally into use in the management of software development, acquisition and utilisation, in the face of substantial evidence of the success of such methods in driving improvements in both quality and productivity.
- We identified ISO 9000 and SEI CMM as two sets of guidelines for setting up a quality system. ISO 9000 series is a standard applicable to a broad spectrum of industries, whereas SEI CMM model is a set of guidelines for setting up a quality system specifically addressing the needs of the software development organisations. Therefore, SEI CMM model addresses various issues pertaining to software industry in a more focussed manner. For example, SEI CMM model suggests a 5-tier structure. On the other hand, ISO 9000 has been formulated by a standards body and therefore the certificate can be used as a contract between externally independent parties, whereas SEI CMM addresses step by step improvements of an organisation's quality practices.
- We discussed the important concepts behind PSP and Six Sigma.

EXERCISES

1. Choose the correct option:
   (a) Which of the following is a practical use of reliability growth modelling? (i) Determine the operational life of an application software
   (ii) Determine when to stop testing
   (iii) Incorporate reliability information while designing
   (iv) Incorporate reliability growth information in the code
   (b) What is the availability of a software with the following reliability figures? Mean Time Between Failure (MTBF) = 25 days, Mean Time To Repair (MTTR) = 6 hours:
   (i) 1 per cent
   (ii) 24 per cent
   (iii) 99 per cent
   (iv) 99.009 per cent
   (c) A software organisation has been assessed at SEI CMM Level 4. Which of the following is a prerequisite to achieve Level 5:
   (i) Defect Detection
   (ii) Defect Prevention

(iii) Defect Isolation
(iv) Defect Propagation

(d) Which one of the following is the focus of modern quality paradigms:
(i) Process assurance
(ii) Product assurance
(iii) Thorough testing
(iv) Thorough testing and rejection of bad products

(e) Which of the following is indicated by the SEI CMM repeatable software development:
(i) Success in development of a software can be repeated
(ii) Success in developmenet of a software can be repeated in related software development projects.
(iii) Success in developmenet of a software can be repeated in all software development projects that the organisation might undertake.
(iv) When the same development team is chosen to develop another software, they can repeat their success.

(f) Which one of the following is the main objective of statistical testing:
(i) Use statistical techniques to design test cases
(ii) Apply statistical techniques to the results of testing to determine if the software has been adequately tested
(iii) Estimate software reliability
(iv) Apply statistical techniques to the results of testing to determine how long testing needs to be carried out

2. Define the terms software reliability and software quality. How can these be measured?
3. Identify the factors which make the measurement of software reliability a much harder problem than the measurement of hardware reliability.
4. Through a simple plot explain how the reliability of a software product changes over its lifetime. Draw the reliability change for a hardware product over its life time and explain why the two plots look so different.
5. What do you understand by a reliability growth model? How is reliability growth modelling useful?
6. Explain using one simple sentence each what you understand by the following reliability measures:
   - A POFOD of 0.001
   - A ROCOF of 0.002
   - MTBF of 200 units
   - Availability of 0.998

7. What is statistical testing? In what way is it useful during software development? Explain in the different steps of statistical testing.
8. Define three metrics to measure software reliability. Do you consider these metrics entirely satisfactory to provide measure of the reliability of a system? Justify your answer.
9. How can you determine the number of latent defects in a software product during the testing phase?
10. State **TRUE** o r **FALSE** of the following. Support your answer with proper reasoning:
    (a) The reliability of a software product increases almost linearly, each time a defect gets detected and fixed.
    (b) As testing continues, the rate of growth of reliability slows down representing a diminishing return of reliability growth with testing effort.
    (c) Modern quality assurance paradigms are centered around carrying out thorough product testing.
    (d) An important use of receiving a ISO 9001 certification by a software organisation is that it can improve its sales efforts by advertising its products as conforming to ISO 9001 certification.
    (e) A highly reliable software can be termed as a good quality software.
    (f) If an organisation assessed at SEI CMM level 1 has developed one software product successfully, then it is expected to repeat its success on similar products.
11. What does the quality parameter "fitness of purpose" mean in the context of software products? Why is this not a satisfactory criterion for determining the quality of software products?
12. Can reliability of a software product be determined by estimating the number of latent defects in the software? If your answer is "yes", explain how reliability can be determined from an estimation of the number of latent defects in a software product. If your answer is "no", explain why can't reliability of a software product be determined from an estimate of the number of latent defects.
13. Why is it important for a software development organisation to obtain ISO 9001 certification?
14. Discuss the relative merits of ISO 9001 certification and the SEI CMM-based quality assessment.
15. List five salient requirements that a software development organisation must comply with before it can be awarded the ISO 9001 certificate.

What are some of the shortcomings of the ISO certification process?

16. With the help of suitable examples discuss the types of software organisations to which ISO 9001, 9002, and 9003 standards respectively are applicable.
17. During software testing process, why is the reliability growth initially high, but slows down later on?
18. If an organisation does not document its quality system, what problems would it face?
19. What according to you is a quality software product?
20. Discuss the stages through which the quality system paradigm and the quality assurance methods have evolved over the years.
21. Which standard is applicable to software industry, ISO 9001, ISO 9002, or ISO 9003?
22. In a software development organisation, identify the persons responsible for carrying out the quality assurance activities. Explain the principal tasks they perform to meet this responsibility.
23. Suppose an organisation mentions in its job advertizement that it has been assessed at level 3 of SEI CMM, what can you infer about the current quality practices at the organisation? What does this organisation have to do to reach SEI CMM level 4?
24. Suppose as the president of a company, you have the choice to either go for ISO 9000 based quality model or SEI CMM based model, which one would you prefer? Give the reasoning behind your choice.
25. What do you understand by total quality management (TQM)? What are the advantages of TQM? Does ISO 9000 standard aim for TQM?
26. What are the principal activities of a modern quality system?
27. In a software development organisation whose responsibility is it to ensure that the products are of high quality? Explain the principal tasks they perform to meet this responsibility.
28. What do you understand by repeatable software development? Organisations assessed at which level SEI CMM maturity achieve repeatable software development?
29. What do you understand by key process area (KPA), in the context of SEI CMM? Would there be any problem if an organisation tries to implement higher level SEI CMM KPAs before achieving lower level KPAs? Justify your answer using suitable examples.
30. What is the Six Sigma quality initiative? To which category of industries is it applicable? Explain the Six Sigma technique adopted by software

organisation with respect to the goal, the procedure, and the outcome.

31. What is the difference between process metrics and product metrics? Give four examples of each.
32. Suppose you want to buy a certain software product and you have kept a purchase precondition that the vendor must install the software, train your manpower on that, and maintain the product for at least a year, only then would you release the payment. Also, you do not foresee any maintenance requirement for the product once it works satisfactorily. Now, you receive bids from three vendors. Two of the vendors quote Rs. 3 lakhs and Rs. 4 lakhs, whereas the third vendor quotes Rs. 10 lakhs saying that the prices would be high because they would be following a good development process as they have been assessed at the Level 5 of SEI CMM. Discuss how you would decide whom to award the contract.
33. A software system is composed of 50 modules. Each module is guaranteed to have a reliability R not less than 0.999. What would be the best case and reliability of the entire system? What should be the reliability of the modules if we require that the system exhibits reliability equal to 0.99999?
34. Explain the importance of software configuration management in modern quality paradigms such as SEI CMM and ISO 9001. An organisation not using any configuration management tool can qualify for which SEI CMM level(s)?
35. List four metrics that can be determined from an analysis of a program's source code and would correlate well with the reliability of the delivered software.
36. Discuss the salient features of the organisational reporting structure of the SQA group as recommended by SEI CMM and ISO 9001. What is the rational behind having such a reporting structure?
37. Suggest two development organisations for whom SEI capability model is not likely to be appropriate. Give reasons why this is the case.
38. What do you understand by Key Process Area (KPA), in the context of SEI CMM?

    Would an organisation encounter any problems, if it tries to implement higher level SEI CMM KPAs before achieving the lower level KPAs? Justify your answer using suitable examples.
39. Can a program be correct and still not exhibit good quality? Explain.
40. What do you understand by defect prevention? Explain how defect prevention can be achieved.

1 To determine the core and non-core parts of a program, you can use a commonly available tool called a *profiler*. On Unix platforms, a tool called "prof" is normally available for this purpose. the reliability figure of a software product is observer-dependent and it is very difficult to absolutely quantify the reliability of the product.

Chapter

12

COMPUTER AIDED SOFTWARE ENGINEERING

In this chapter, we will discuss about computer aided software engineering (CASE) and how use of CASE tools help to improve software development effort and maintenance effort. Of late, CASE has emerged as a much talked topic in software industries. Software is becoming the costliest component in any computer installation. Even though hardware prices keep dropping like never and falling below even the most optimistic expectations, software prices are becoming costlier due to increased manpower costs. This scenario has got most managers worried. In this scene, CASE tools promise effort and cost reduction in software development and maintenance. Therefore, deployment and development of CASE tools have become pet subjects for most software project managers. For software engineers, CASE tools promises to take drudgery out of routine jobs, and help develop better quality products more efficiently.

With this brief introduction and motivation for studying CASE tools, we will first define the scope of CASE and examine the different concepts associated with CASE. Subsequently, we will discuss the features of different types of CASE tools.

12.1 CASE AND ITS SCOPE

We first need to define what is a CASE tool and what is a CASE environment. A CASE tool is a generic term used to denote any form of automated support for software engineering, In a more restrictive sense a CASE tool can mean any tool used to automate some activity associated with software development. Many CASE tools are now available. Some of these tools assist in phase-related tasks such as

specification, structured analysis, design, coding, testing, etc. and others to non-phase activities such as project management and configuration management. The primary objectives in using any CASE tool are:

- To increase productivity.
- To help produce better quality software at lower cost.

12.2 CASE ENVIRONMENT

Although individual CASE tools are useful, the true power of a tool set can be realised only when these set of tools are integrated into a common framework or environment. If the different CASE tools are not integrated, then the data generated by one tool would have to input to the other tools. This may also involve format conversions as the tools developed by different vendors are likely to use different formats. This results in additional effort of exporting data from one tool and importing to another. Also, many tools do not allow exporting data and maintain the data in proprietary formats.

CASE tools are characterised by the stage or stages of software development life cycle on which they focus. Since different tools covering different stages share common information, it is required that they integrate through some central repository to have a consistent view of information associated with the software. This central repository is usually a data dictionary containing the definition of all composite and elementary data items. Through the central repository all the CASE tools in a CASE environment share common information among themselves. Thus a CASE environment facilitates the automation of the step-by-step methodologies for software development. In contrast to a CASE environment, a programming environment is an integrated collection of tools to support only the coding phase of software development. The tools commonly integrated in a programming environment are a text editor, a compiler, and a debugger. The different tools are integrated to the extent that once the compiler detects an error, the editor takes automatically goes to the statements in error and the error statements are highlighted. Examples of popular programming environments are Turbo C environment, Visual Basic, Visual C++, etc. A schematic representation of a CASE environment is shown in Figure 12.1.

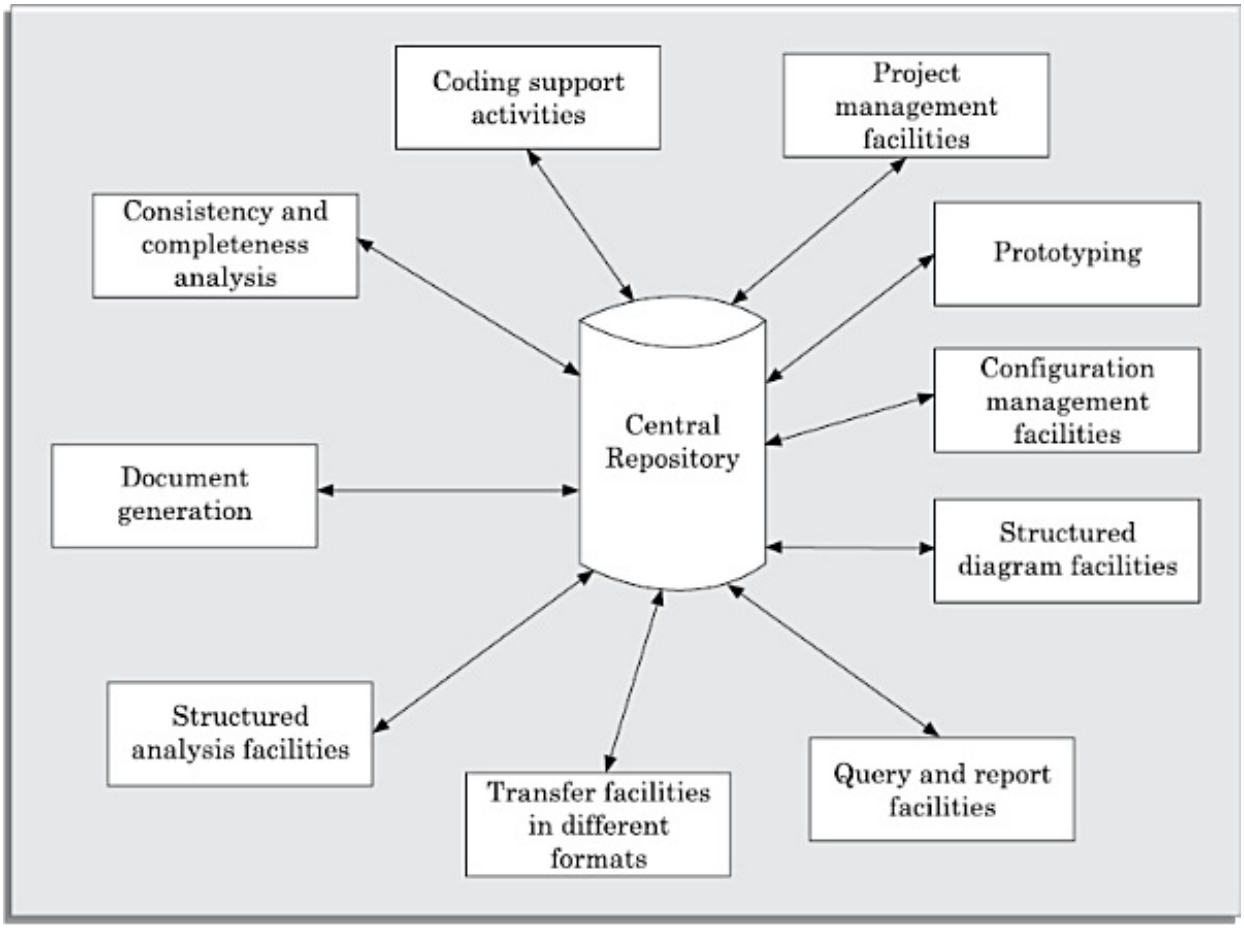

Figure 12.1: A CASE environment.

The standard programming environments such as Turbo C, Visual C++, etc. come equipped with a program editor, compiler, debugger, linker, etc., All these tools are integrated. If you click on an error reported by the compiler, not only does it take you into the editor, but also takes the cursor to the specific line or statement causing the error.

12.2.1 Benefits of CASE

Several benefits accrue from the use of a CASE environment or even isolated CASE tools. Let us examine some of these benefits:

- A key benefit arising out of the use of a CASE environment is cost saving through all developmental phases. Different studies carry out to measure the impact of CASE, put the effort reduction between 30 per cent and 40 per cent.
- Use of CASE tools leads to considerable improvements in quality. This is mainly due to the facts that one can effortlessly iterate through the different phases of software development, and the chances of human error is considerably reduced.

- CASE tools help produce high quality and consistent documents. Since the important data relating to a software product are maintained in a central repository, redundancy in the stored data is reduced, and therefore, chances of inconsistent documentation is reduced to a great extent.
- CASE tools take out most of the drudgery in a software engineers work. For example, they need not check meticulously the balancing of the DFDs, but can do it effortlessly through the press of a button.
- CASE tools have led to revolutionary cost saving in software maintenance efforts. This arises not only due to the tremendous value of a CASE environment in traceability and consistency checks, but also due to the systematic information capture during the various phases of software development as a result of adhering to a CASE environment.
- Introduction of a CASE environment has an impact on the style of working of a company, and makes it oriented towards the structured and orderly approach.

12.3 CASE SUPPORT IN SOFTWARE LIFE CYCLE

Let us examine the various types of support that CASE provides during the different phases of a software life cycle. CASE tools should support a development methodology, help enforce the same, and provide certain amount of consistency checking between different phases. Some of the possible support that CASE tools usually provide in the software development life cycle are discussed below.

12.3.1 Prototyping Support

We have already seen that prototyping is useful to understand the requirements of complex software products, to demonstrate a concept, to market new ideas, and so on. The prototyping CASE tool's requirements are as follows:

- Define user interaction.
- Define the system control flow.
- Store and retrieve data required by the system.
- Incorporate some processing logic.

There are several stand alone prototyping tools. But a tool that integrates with the data dictionary can make use of the entries in the data dictionary,

help in populating the data dictionary and ensure the consistency between the design data and the prototype.

A good prototyping tool should support the following features:

- Since one of the main uses of a prototyping CASE tool is graphical user interface (GUI) development, a prototyping CASE tool should support the user to create a GUI using a graphics editor. The user should be allowed to define all data entry forms, menus and controls.
- It should integrate with the data dictionary of a CASE environment.
- If possible, it should be able to integrate with external user defined modules written in C or some popular high level programming languages.
- The user should be able to define the sequence of states through which a created prototype can run. The user should also be allowed to control the running of the prototype.
- The run time system of prototype should support mock up run of the actual system and management of the input and output data.

12.3.2 Structured Analysis and Design

Several diagramming techniques are used for structured analysis and structured design. A CASE tool should support one or more of the structured analysis and design technique. The CASE tool should support effortlessly drawing analysis and design diagrams. The CASE tool should support drawing fairly complex diagrams and preferably through a hierarchy of levels. It should provide easy navigation through different levels and through design and analysis. The tool must support completeness and consistency checking across the design and analysis and through all levels of analysis hierarchy. Wherever it is possible, the system should disallow any inconsistent operation, but it may be very difficult to implement such a feature. Whenever there is heavy computational load while consistency checking, it should be possible to temporarily disable consistency checking.

12.3.3 Code Generation

As far as code generation is concerned, the general expectation from a CASE tool is quite low. A reasonable requirement is traceability from source file to design data. More pragmatic support expected from a CASE tool during code generation phase are the following:

- The CASE tool should support generation of module skeletons or templates in one or more popular languages. It should be possible to include copyright message, brief description of the module, author name and the date of creation in some selectable format.
- The tool should generate records, structures, class definition automatically from the contents of the data dictionary in one or more popular programming languages.
- It should generate database tables for relational database management systems.
- The tool should generate code for user interface from prototype definition for X window and MS window based applications.

12.3.4 Test Case Generator

The CASE tool for test case generation should have the following features:

- It should support both design and requirement testing
- It should generate test set reports in ASCII format which can be directly imported into the test plan document.

12.4 OTHER CHARACTERISTICS OF CASE TOOLS

The characteristics listed in this section are not central to the functionality of CASE tools but significantly enhance the effectivity and usefulness of CASE tools.

12.4.1 Hardware and Environmental Requirements

In most cases, it is the existing hardware that would place constraints upon the CASE tool selection. Thus, instead of defining hardware requirements for a CASE tool, the task at hand becomes to fit in an optimal configuration of CASE tool in the existing hardware capabilities. Therefore, we have to emphasise on selecting the most optimal CASE tool configuration for a given hardware configuration.

The heterogeneous network is one instance of distributed environment and we choose this for illustration as it is more popular due to its machine independent features. The CASE tool implementation in heterogeneous network makes use of client-server paradigm. The multiple clients which run different modules access data dictionary through this server. The data

dictionary server may support one or more projects. Though it is possible to run many servers for different projects but distributed implementation of data dictionary is not common. The tool set is integrated through the data dictionary which supports multiple projects, multiple users working simultaneously and allows to share information between users and projects. The data dictionary provides consistent view of all project entities, e.g., a data record definition and its entity-relationship diagram be consistent. The server should depict the per-project logical view of the data dictionary. This means that it should allow back up/restore, copy, cleaning part of the data dictionary, etc. The tool should work satisfactorily for maximum possible number of users working simultaneously. The tool should support multi-windowing environment for the users. This is important to enable the users to see more than one diagram at a time. It also facilitates navigation and switching from one part to the other.

12.4.2 Documentation Support

The deliverable documents should be organized graphically and should be able to incorporate text and diagrams from the central repository. This helps in producing up-to-date documentation. The CASE tool should integrate with one or more of the commercially available desk-top publishing packages. It should be possible to export text, graphics, tables, data dictionary reports to the DTP package in standard forms such as PostScript.

12.4.3 Pro ject Management

It should support collecting, storing, and analysing information on the software project's progress such as the estimated task duration, scheduled and actual task start, completion date, dates and results of the reviews, etc.

12.4.4 External Interface

The tool should allow exchange of information for reusability of design. The information which is to be exported by the tool should be preferably in ASCII format and support open architecture. Similarly, the data dictionary should provide a programming interface to access information. It is required for integration of custom utilities, building new techniques, or populating the data dictionary.

12.4.5 Reverse Engineering Support

The tool should support generation of structure charts and data dictionaries from the existing source codes. It should populate the data dictionary from the source code. If the tool is used for re-engineering information systems, it should contain conversion tool from indexed sequential file structure, hierarchical and network database to relational database systems.

12.4.6 Data Dictionary Interface

The data dictionary interface should provide view and update access to the entities and relations stored in it. It should have print facility to obtain hard copy of the viewed screens. It should provide analysis reports like cross-referencing, impact analysis, etc. Ideally, it should support a query language to view its contents.

12.4.7 Tutorial and Help

The application of CASE tool and thereby its success depends on the users' capability to effectively use all the features supported. Therefore, for the uninitiated users, a tutorial is very important. The tutorial should not be limited to teaching the user interface part only, but should comprehensively cover the following points:

- The tutorial should cover all techniques and facilities through logically classified sections.
- The tutorial should be supported by proper documentation.

12.5 TOWARDS SECOND GENERATION CASE TOOL

An important feature of the second generation CASE tool is the direct support of any adapted methodology. This would necessitate the function of a CASE administrator for every organisation, who can tailor the CASE tool to a particular methodology. In addition, we may look forward to the following features in the second generation CASE tool:

Intelligent diagramming support: The fact that diagramming techniques are useful for system analysis and design is well established. The future CASE tools would provide help to aesthetically and automatically layout the diagrams.

Integration with implementation environment: The CASE tools should

provide integration between design and implementation.

Data dictionary standards: The user should be allowed to integrate many development tools into one environment. It is highly unlikely that any one vendor will be able to deliver a total solution. Moreover, a preferred tool would require tuning up for a particular system. Thus the user would act as a system integrator. This is possible only if some standard on data dictionary emerges.

Customisation support: The user should be allowed to define new types of objects and connections. This facility may be used to build some special methodologies. Ideally it should be possible to specify the rules of a methodology to a rule engine for carrying out the necessary consistency checks.

12.6 ARCHITECTURE OF A CASE ENVIRONMENT

The architecture of a typical modern CASE environment is shown diagrammatically in Figure 12.2. The important components of a modern CASE environment are user interface, tool set, object management system (OMS), and a repository. We have already seen the characteristics of the tool set. Let us examine the other components of a CASE environment.

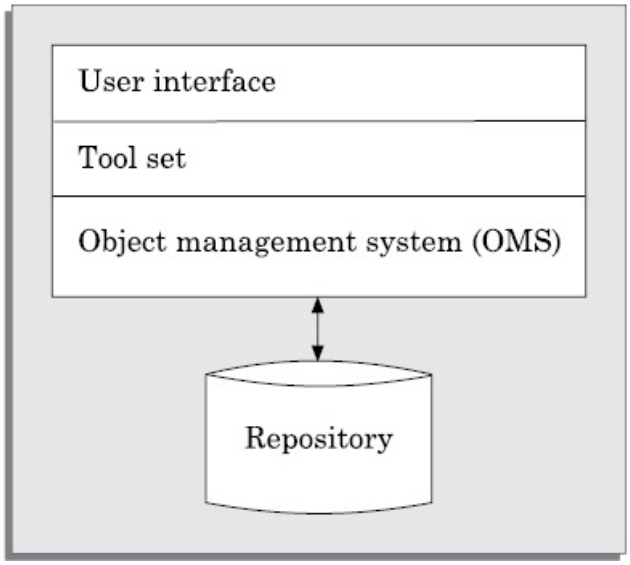

Figure 12.2: Architecture of a modern CASE environment.

User interface

The user interface provides a consistent framework for accessing the different tools thus making it easier for the users to interact with the

different tools and reducing the overhead of learning how the different tools are used.

Object management system and repository

Different case tools represent the software product as a set of entities such as specification, design, text data, project plan, etc. The object management system maps these logical entities into the underlying storage management system (repository). The commercial relational database management systems are geared towards supporting large volumes of information structured as simple relatively short records. There are a few types of entities but large number of instances. By contrast, CASE tools create a large number of entity and relation types with perhaps a few instances of each. Thus the object management system takes care of appropriately mapping these entities into the underlying storage management system.

SUMMARY

- We have highlighted some of the important features of the present day CASE tools and have discussed the emerging trends.
- Use of CASE tools is becoming almost indispensable for large software projects where a team of software engineers work together. The trend is now towards distributed workstation-based CASE tools.
- We pointed out some of the desirable features of the distributed workstation-based CASE tools.

EXERCISES

1. Choose the correct option:
   (a) Which one of the following effectively integrates the different tools in a CASE environment?
   (i) Software requirements specification (SRS) document
   (ii) Central data repository
   (iii) Incremental compilation
   (iv) User intervention
   (b) Which one of the following CASE tools is usually not part of a programming environment?
   (i) Compiler
   (ii) debugger

(iii) Modelling tool
(iv) Editor

(c) Which of the following CASE tools is usually not useful during a corrective maintenance activity?
(i) Regression test selection tool
(ii) Reverse engineering tool
(iii) Symbolic debugger
(iv) Requirements capture tool

2. What do you understand by the terms a CASE tool and a CASE environment? Why integration tools increases the power of the tools? Explain using some examples.
3. What is a programming environment?
4. What are the main advantages of using CASE tools?
5. What are some of the important features that a future generation CASE tool should support?
6. Identify the CASE support that can be availed of during a large maintenance effort concerning a large legacy software.
7. Discuss the role of the data dictionary in a CASE environment.
8. Schematically draw the architecture of a CASE environment and explain how the different tools are integrated.

Chapter
13

SOFTWARE MAINTENANCE

Many students and practising engineers have a preconceived bias against software maintenanc e work. The mention of the word maintenance brings up the image of a screw driver, wielding mechanic with soiled hands holding onto a bagful of spare parts. It would be the objective of this chapter to clear up this misnomer, provide some intuitive understanding of the software maintenance projects, and to familiarise you with the latest techniques in software maintenance.

Software maintenance denotes any changes made to a software product after it has been delivered to the customer. Maintenance is inevitable for almost any kind of product. However, most products need maintenance due to the wear and tear caused by use. On the other hand, software products do not need maintenance on this count, but need maintenance to correct errors, enhance features, port to new platforms, etc.

In Section 13.1, we examine some general issues concerning maintenance projects. In Section 13.2, we discuss some basic ideas about software reverse engineering. In Section 13.3 we discuss two software maintenance process models which attempt to systematise the software development effort and finally we discuss some concepts involved in cost estimation of maintenance efforts.

13.1 CHARACTERISTICS OF SOFTWARE MAINTENANCE

In this section, we first classify the different maintenance efforts into a few classes. Next, we discuss some general characteristics of the maintenance projects. We also discuss some special problems associated with maintenance projects.

Software maintenance is becoming an important activity of a large number of organisations. This is no surprise, given the rate of hardware obsolescence, the immortality of a software product per se, and the demand of the user

community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features. When the hardware platform changes, and a software product performs some low-level functions, maintenance is necessary. Also, whenever the support environment of a software product changes, the software product requires rework to cope up with the newer interface. For instance, a software product may need to be maintained when the operating system changes. Thus, every software product continues to evolve after its development through maintenance efforts.

Types of Software Maintenance

There are three types of software maintenance, which are described as follows:

Corrective: Corrective maintenance of a software product is necessary either to rectify the bugs observed while the system is in use.

Adaptive: A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.

Perfective: A software product needs maintenance to support the new features that users want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

13.1.1 Characteristics of Software Evolution

Lehman and Belady have studied the characteristics of evolution of several software products [1980]. They have expressed their observations in the form of laws. Their important laws are presented in the following subsection. But a word of caution here is that these are generalisations and may not be applicable to specific cases and also most of these observations concern large software projects and may not be appropriate for the maintenance and evolution of very small products.

Lehman's first law: A software product must change continually or become progressively less useful. Every software product continues to evolve after its development through maintenance efforts. Larger products stay in operation for longer times because of higher replacement costs and therefore tend to incur higher maintenance efforts. This law clearly shows that every product

irrespective of how well designed must undergo maintenance. In fact, when a product does not need any more maintenance, it is a sign that the product is about to be retired/discarded. This is in contrast to the common intuition that only badly designed products need maintenance. In fact, good products are maintained and bad products are thrown away.

Lehman's second law: The structure of a program tends to degrade as more and more maintenance is carried out on it. The reason for the degraded structure is that when you add a function during maintenance, you build on top of an existing program, often in a way that the existing program was not intended to support. If you do not redesign the system, the additions will be more complex that they should be. Due to quick-fix solutions, in addition to degradation of structure, the documentations become inconsistent and become less helpful as more and more maintenance is carried out.

Lehman's third law: Over a program's lifetime, its rate of development is approximately constant. The rate of development can be quantified in terms of the lines of code written or modified. Therefore this law states that the rate at which code is written or modified is approximately the same during development and maintenance.

13.1.2 Special Problems Associated with Software Maintenance

Software maintenance work currently is typically much more expensive than what it should be and takes more time than required. The reasons for this situation are the following:

Software maintenance work in organisations is mostly carried out using ad hoc techniques. The primary reason being that software maintenance is one of the most neglected areas of software engineering. Even though software maintenance is fast becoming an important area of work for many companies as the software products of yester years age, still software maintenance is mostly being carried out as fire-fighting operations, rather than through systematic and planned activities.

Software maintenance has a very poor image in industry. Therefore, an organisation often cannot employ bright engineers to carry out maintenance work. Even though maintenance suffers from a poor image, the work involved is often more challenging than development work. During maintenance it is necessary to thoroughly understand someone else's work, and then carry out the required modifications and extensions.

Another problem associated with maintenance work is that the majority of

software products needing maintenance are legacy products. Though the word legacy implies "aged" software, but there is no agreement on what exactly is a legacy system. It is prudent to define a legacy system as any software system that is hard to maintain. The typical problems associated with legacy systems are poor documentation, unstructured (spaghetti code with ugly control structure), and lack of personnel knowledgeable in the product. Many of the legacy systems were developed long time back. But, it is possible that a recently developed system having poor design and documentation can be considered to be a legacy system.

13.2 SOFTWARE REVERSE ENGINEERING

Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code. The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system. Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured. Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.

The first stage of reverse engineering usually focuses on carrying out cosmetic changes to the code to improve its readability, structure, and understandability, without changing any of its functionalities. A way to carry out these cosmetic changes is shown schematically in Figure 13.1. A program can be reformatted using any of the several available prettyprinter programs which layout the program neatly. Many legacy software products are difficult to comprehend with complex control structure and unthoughtful variable names. Assigning meaningful variable names is important because we had seen in Chapter 9 that meaningful variable names is the most helpful code documentation. All variables, data structures, and functions should be assigned meaningful names wherever possible. Complex nested conditionals in the program can be replaced by simpler conditional statements or whenever appropriate by case statements.

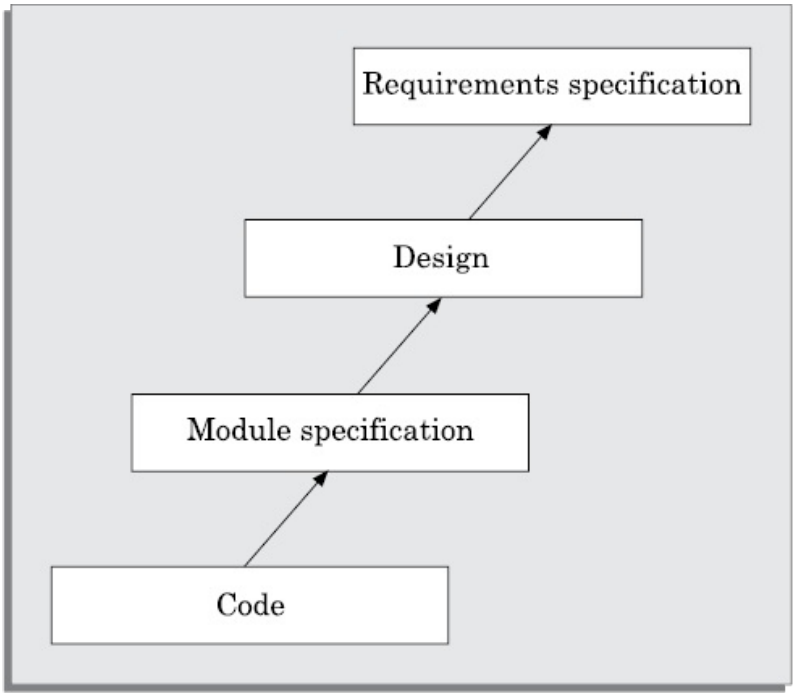

Figure 13.1: A process model for reverse engineering.

After the cosmetic changes have been carried out on a legacy software, the proces of extracting the code, design, and the requirements specification can begin. These activities are schematically shown in Figure 13.2. In order to extract the design, a full understanding of the code is needed. Some automatic tools can be used to derive the data flow and control flow diagram from the code. The structure chart (module invocation sequence and data interchange among modules) should also be extracted. The SRS document can be written once the full code has been thoroughly understood and the design extracted.

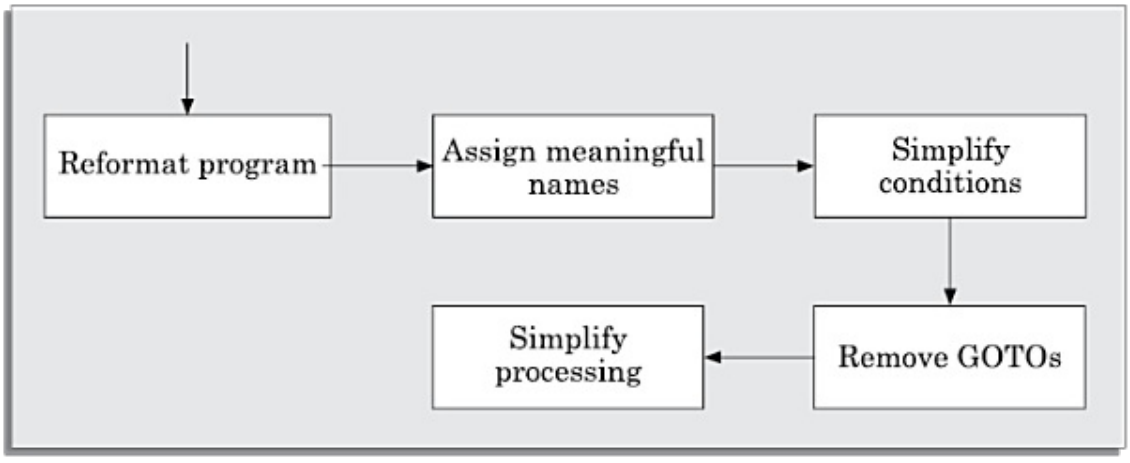

Figure 13.2: Cosmetic changes carried out before reverse engineering.

13.3 SOFTWARE MAINTENANCE PROCESS MODELS

Before discussing process models for software maintenance, we need to analyse various activities involved in a typical software maintenance project. The activities involved in a software maintenance project are not unique and depend on several factors such as: (i) the extent of modification to the product required, (ii) the resources available to the maintenance team, (iii) the conditions of the existing product (e.g., how structured it is, how well documented it is, etc.), (iii) the expected project risks, etc. When the changes needed to a software product are minor and straightforward, the code can be directly modified and the changes appropriately reflected in all the documents.

However, more elaborate activities are required when the required changes are not so trivial. Usually, for complex maintenance projects for legacy systems, the software process can be represented by a reverse engineering cycle followed by a forward engineering cycle with an emphasis on as much reuse as possible from the existing code and other documents.

Since the scope (activities required) for different maintenance projects vary widely, no single maintenance process model can be developed to suit every kind of maintenance project. However, two broad categories of process models can be proposed.

First model

The first model is preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant documents later. This maintenance process is graphically presented in Figure 13.3. In this approach, the project starts by gathering the requirements for changes. The requirements are next analysed to formulate the strategies to be adopted for code change. At this stage, the association of at least a few members of the original development team goes a long way in reducing the cycle time, especially for projects involving unstructured and inadequately documented code. The availability of a working old system to the maintenance engineers at the maintenance site greatly facilitates the task of the maintenance team as they get a good insight into the working of the old system and also can compare the working of their modified system with the old system. Also, debugging of the re-engineered system becomes easier as the program traces of both the systems can be compared to localise the bugs.

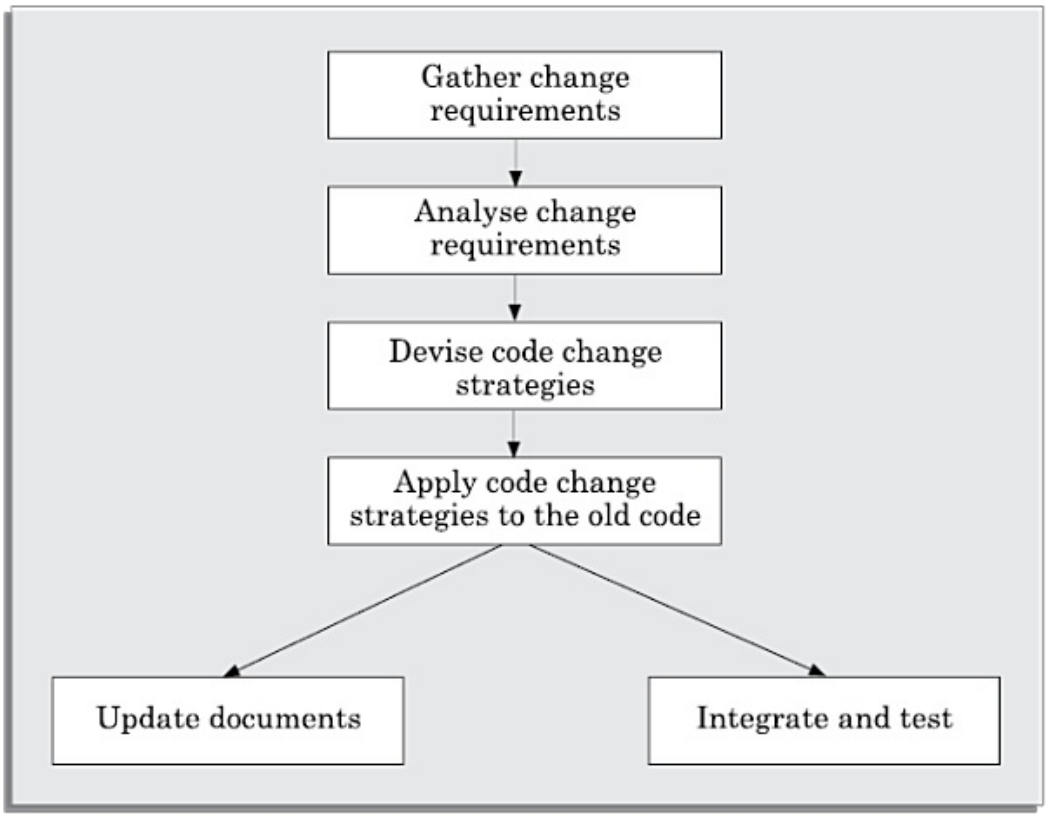

Figure 13.3: Maintenance process model 1.

Second model

The second model is preferred for projects where the amount of rework required is significant. This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. Such an approach is also known as software re-engineering. This process model is depicted in Figure 13.4.

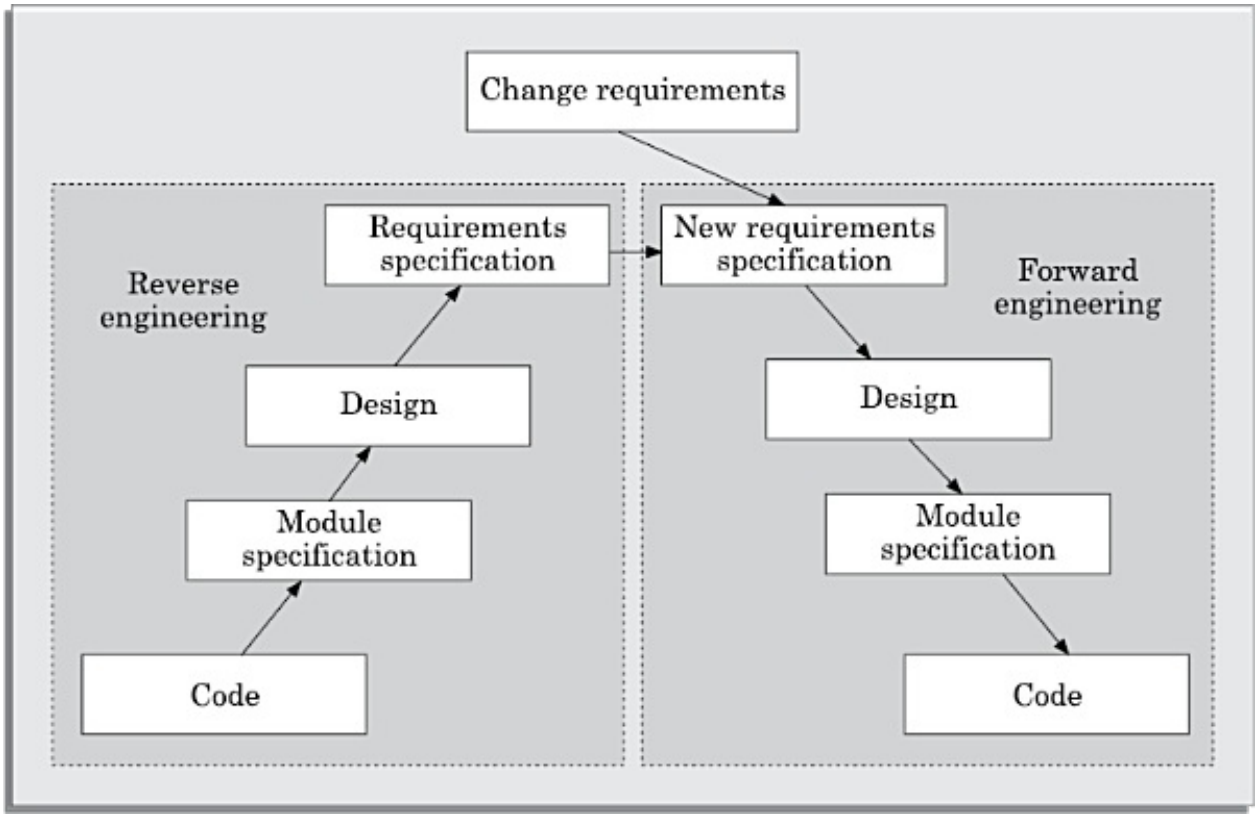

Figure 13.4: Maintenance process model 2.

The reverse engineering cycle is required for legacy products. During the reverse engineering, the old code is analysed (abstracted) to extract the module specifications. The module specifications are then analysed to produce the design. The design is analysed (abstracted) to produce the original requirements specification. The change requests are then applied to this requirements specification to arrive at the new requirements specification. At this point a forward engineering is carried out to produce the new code. At the design, module specification, and coding a substantial reuse is made from the reverse engineered products. An important advantage of this approach is that it produces a more structured design compared to what the original product had, produces good documentation, and very often results in increased efficiency. The efficiency improvements are brought about by a more efficient design. However, this approach is more costly than the first approach. An empirical study indicates that process 1 is preferable when the amount of rework is no more than 15 per cent (see Figure 13.5).

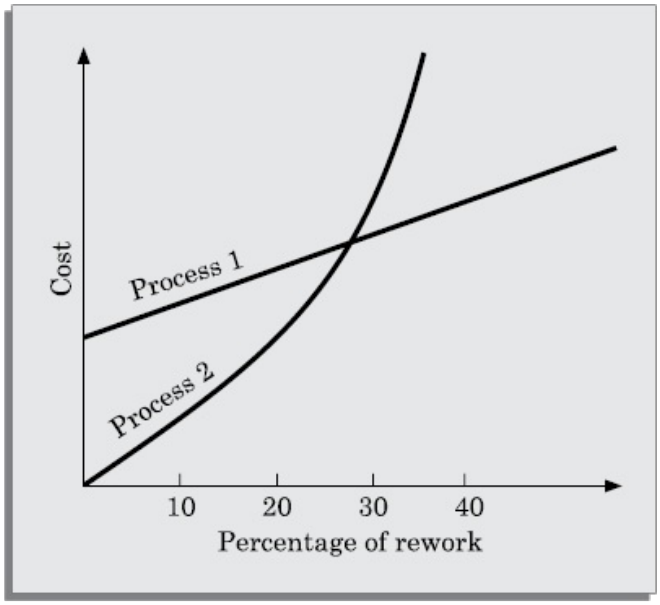

Figure 13.5: Empirical estimation of maintenance cost versus percentage rework.

Besides the amount of rework, several other factors might affect the decision regarding using process model 1 over process model 2 as follows:

- Re-engineering might be preferable for products which exhibit a high failure rate.
- Re-engineering might also be preferable for legacy products having poor design and code structure.

13.4 ESTIMATION OF MAINTENANCE COST

We had earlier pointed out that maintenance efforts require about 60 per cent of the total life cycle cost for a typical software product. However, maintenance costs vary widely from one application domain to another. For embedded systems, the maintenance cost can be as much as 2 to 4 times the development cost.

Boehm [1981] proposed a formula for estimating maintenance costs as part of his COCOMO cost estimation model. Boehm's maintenance cost estimation is made in terms of a quantity called the annual change traffic (ACT). Boehm defined ACT as the fraction of a software product's source instructions which undergo change during a typical year either through addition or deletion.

$$\text{ACT} = \frac{\text{KLOC}_{\text{added}} + \text{KLOC}_{\text{deleted}}}{\text{KLOC}_{\text{total}}}$$

where, KLOCadded is the total kilo lines of source code added during maintenance. KLOCdeleted is the total KLOC deleted during

maintenance. Thus, the code that is changed, should be counted in both the code added and code deleted.

The annual change traffic (ACT) is multiplied with the total development cost to arrive at the maintenance cost:

$$\text{Maintenance cost} = \text{ACT} \times \text{Development cost}$$

Most maintenance cost estimation models, however, give only approximate results because they do not take into account several factors such as experience level of the engineers, and familiarity of the engineers with the product, hardware requirements, software complexity, etc.

SUMMARY

- In this chapter, we discussed some fundamental concepts associated with software maintenance activities.
- Maintenance is the most expensive phase of the software life cycle and therefore it is usually cost-effective to invest in time and effort while developing the product and to emphasise on maintainability of the product to reduce the maintenance costs.
- We discussed the activities in reverse engineering and then discussed two maintenance process models. We also discussed the applicability of these two process models to maintenance projects.
- We highlighted the salient points in costing maintenance projects.

EXERCISES

1. Choose the correct option:
   (a) Which of the following is not a cause for software maintenance for a typical product?
   (i) It is not possible to guarantee that a software is defect-free even after thorough
   testing.
   (ii) The deployment platform may change over time.
   (iii) The user's needs may change over time.
   (iv) Software undergoes wear and tear after long usage.
   (b) A legacy software product refers to a software that is:
   (i) Developed at least 50 years ago. (ii) Obsolete software product.
   (iii) Software product that has poor design structure and code.
   (iv) Software product that could not be tested properly before product

delivery.

(c) Which of the following assertions is true?

(i) Legacy products automatically imply very old products.

(ii) The total effort spent in maintaining an average product typically exceeds the effort in developing it.

(iii) Reverse engineering encompasses re-engineering.

(iv) Reeingineering encompasses reverse engineering.

(d) Which of the following types of maintenance consumes the maximum effort for a typical software?

(i) Adaptive

(ii) Corrective

(iii) Preventive

(iv) Perfective

2. What are the different types of maintenance that a software product might need? Why are these maintenance required?

3. Explain why every software system must undergo maintenance or progressively become less useful.

4. Discuss the process models for software maintenance and indicate how you would select an appropriate maintenance model for a maintenance project at hand.

5. State whether the following statements are **TRUE** or **FALSE**. Give reasons for your answer.

(a) Legacy software products are those products which have been developed long time back.

(b) Corrective maintenance is the type of maintenance that is most frequently carried out on a typical software product.

6. What do you mean by the term software reverse engineering? Why is it required? Explain the different activites undertaken during reverse engineering.

7. What do you mean by the term software re-engineering? Why is it required? Explain the different activities undertaken during reverse engineering.

8. If a software product costed Rs. 10,000,000 for development, compute the annual maintenance cost given that every year approximately 5 per cent of the code needs modification. Identify the factors which render the maintenance cost estimation inaccurate?

9. What is a legacy software product? Explain the problems one would encounter while maintaining a legacy product.

Chapter 14

SOFTWARE REUSE

Software products are expensive. Therefore, software project managers are always worried about the high cost of software development and are desperately looking for ways to cut development cost. A possible way to reduce development cost is to reuse parts from previously developed software. In addition to reduced development cost and time, reuse also leads to higher quality of the developed products since the reusable components are ensured to have high quality. A reuse approach that is of late gaining prominence is component-based development. Component-based software development is different from the traditional software development in the sense that software is developed by assembling software from off-the-shelf components.

Software development with reuse is very similar to a modern hardware engineer building an electronic circuit by using standard types of ICs and other components. In this Chapter, we will review the state of art in software reuse.

14.1 WHAT CAN BE REUSED?

Before discussing the details of reuse techniques, it is important to deliberate about the kinds of the artifacts associated with software development that can be reused. Almost all artifacts associated with software development, including project plan and test plan can be reused. However, the prominent items that can be effectively reused are:

- Requirements specification
- Design
- Code
- Test cases

- Knowledge

Knowledge is the most abstract development artifact that can be reused. Out of all the reuse artifacts, reuse of knowledge occurs automatically without any conscious effort in this direction. However, two major difficulties with unplanned reuse of knowledge is that a developer experienced in one type of product might be included in a team developing a different type of software. Also, it is difficult to remember the details of the potentially reusable development knowledge. A planned reuse of knowledge can increase the effectiveness of reuse. For this, the reusable knowledge should be systematically extracted and documented. But, it is usually very difficult to extract and document reusable knowledge.

14.2 WHY ALMOST NO REUSE SO FAR?

A common scenario in many software development industries is explained further. Engineers working in software development organisations often have a feeling that the current system that they are developing is similar to the last few systems built. However, no attention is paid on how not to duplicate what can be reused from previously developed systems. Everything is being built from scratch. The current system falls behind schedule and no one has time to figure out how the similarity between the current system and the systems developed in the past can be exploited.

Even those organisations which embark on a reuse program, in spite of the above difficulty, face other problems. Creation of components that are reusable in different applications is a difficult problem. It is very difficult to anticipate the exact components that can be reused across different applications. But, even when the reusable components are carefully created and made available for reuse, programmers prefer to create their own, because the available components are difficult to understand and adapt to the new applications.

In this context, the following observation is significant: The routines of mathematical libraries are being reused very successfully by almost every programmer. No one in their mind would think of writing a routine to compute sine or cosine. Let us investigate why reuse of commonly used mathematical functions is so easy. Several interesting aspects emerge. Cosine means the same to all. Everyone has clear ideas about what kind of argument should cosine take, the type of processing to be carried out and the results returned.

Secondly, mathematical libraries have a small interface. For example, cosine requires only one parameter. Also, the data formats of the parameters are standardised. These are some fundamental issues which would remain valid for all our subsequent discussions on reuse. In the following section, we discuss the issues that must be addressed while starting any reuse program in an organisation.

14.3 BASIC ISSUES IN ANY REUSE PROGRAM

The following are some of the basic issues that must be clearly understood for starting any reuse program:

- Component creation.
- Component indexing and storing.
- Component search.
- Component understanding.
- Component adaptation.
- Repository maintenance.

Component creation: For component creation, the reusable components have to be first identified. Selection of the right kind of components having potential for reuse is important. In Section 14.4, we discuss domain analysis as a promising technique which can be used to create reusable components.

Component indexing and storing

Indexing requires classification of the reusable components so that they can be easily searched when we look for a component for reuse. The components need to be stored in a *relational database management system* (RDBMS) or an *object-oriented database system* (ODBMS) for efficient access when the number of components becomes large.

Component searching

The programmers need to search for right components matching their requirements in a database of components. To be able to search components efficiently, the programmers require a proper method to describe the components that they are looking for.

Component understanding

The programmers need a precise and sufficiently complete

understanding of what the component does to be able to decide whether they can reuse the component. To facilitate understanding, the components should be well documented and should do something simple.

Component adaptation

Often, the components may need adaptation before they can be reused, since a selected component may not exactly fit the problem at hand. However, tinkering with the code is also not a satisfactory solution because this is very likely to be a source of bugs.

Repository maintenance

A component repository once is created requires continuous maintenance. New components, as and when created have to be entered into the repository. The faulty components have to be tracked. Further, when new applications emerge, the older applications become obsolete. In this case, the obsolete components might have to be removed from the repository.

14.4 A REUSE APPROACH

A promising approach that is being adopted by many organisations is to introduce a building block approach into the software development process. For this, the reusable components need to be identified after every development project is completed. The reusability of the identified components has to be enhanced and these have to be cataloged into a component library. It must be clearly understood that an issue crucial to every reuse effort is the identification of reusable components. Domain analysis is a promising approach to identify reusable components. In the following subsections, we discuss the domain analysis approach to create reusable components.

14.4.1 Domain Analysis

The aim of domain analysis is to identify the reusable components for a problem domain.

Reuse domain

A reuse domain is a technically related set of application areas. A body of information is considered to be a problem domain for reuse, if a deep

and comprehensive relationship exists among the information items as characterised by patterns of similarity among the development components of the software product. A reuse domain is a shared understanding of some community, characterised by concepts, techniques, and terminologies that show some coherence. Examples of domains are accounting software domain, banking software domain, business software domain, manufacturing automation software domain, telecommunication software domain, etc.

Just to become familiar with the vocabulary of a domain requires months of interaction with the experts. Often, one needs to be familiar with a network of related domains for successfully carrying out domain analysis. Domain analysis identifies the objects, operations, and the relationships among them. For example, consider the airline reservation system, the reusable objects can be seats, flights, airports, crew, meal orders, etc. The reusable operations can be scheduling a flight, reserving a seat, assigning crew to flights, etc. We can see that the domain analysis generalises the application domain. A domain model transcends specific applications. The common characteristics or the similarities between systems are generalised.

During domain analysis, a specific community of software developers get together to discuss community-wide solutions. Analysis of the application domain is required to identify the reusable components. The actual construction of the reusable components for a domain is called *domain engineering*.

Evolution of a reuse domain

The ultimate results of domain analysis is development of problem-oriented languages. The problem-oriented languages are also known as *application generators*. These application generators, once developed form application development standards. The domains slowly develop. As a domain develops, we may distinguish the various stages it undergoes:

Stage 1 : There is no clear and consistent set of notations. Obviously, no reusable components are available. All software is written from scratch.

Stage 2 : Here, only experience from similar projects are used in a development effort. This means that there is only knowledge reuse.

Stage 3: At this stage, the domain is ripe for reuse. The set of concepts are stabilised and the notations standardised. Standard solutions to standard

problems are available. There is both knowledge and component reuse.

Stage 4: The domain has been fully explored. The software development for the domain can largely be automated. Programs are not written in the traditional sense any more. Programs are written using a domain specific language, which is also known as an *application generator*.

14.4.2 Component Classification

Components need to be properly classified in order to develop an effective indexing and storage scheme. We have already remarked that hardware reuse has been very successful. If we look at the classification of hardware components for clue, then we can observe that hardware components are classified using a multilevel hierarchy. At the lowest level, the components are described in several forms—natural language description, logic schema, timing information, etc. The higher the level at which a component is described, the more is the ambiguity. This has motivated the Prieto-Diaz's classification scheme.

Prieto-Diaz's classification scheme

Each component is best described using a number of different characteristics or facets. For example, objects can be classified using the following:

- Actions they embody.
- Objects they manipulate.
- Data structures used.
- Systems they are part of, etc.

Prieto-Diaz's faceted classification scheme requires choosing an n-tuple that best fits a component. Faceted classification has advantages over enumerative classification. Strictly enumerative schemes use a pre-defined hierarchy. Therefore, these force you to search for an item that best fits the component to be classified. This makes it very difficult to search a required component. Though cross referencing to other items can be included, the resulting network becomes complicated.

14.4.3 Searching

The domain repository may contain thousands of reuse items. In such large domains, what is the most efficient way to search an item that

one is looking for? A popular search technique that has proved to be very effective is one that provides a web interface to the repository. Using such a web interface, one would search an item using an approximate automated search using key words, and then from these results would do a browsing using the links provided to look up related items. The approximate automated search locates products that appear to fulfill some of the specified requirements. The items located through the approximate search serve as a starting point for browsing the repository. These serve as the starting point for browsing the repository. The developer may follow links to other products until a sufficiently good match is found. Browsing is done using the keyword-to-keyword, keyword-to-product, and product- to-product links. These links help to locate additional products and compare their detailed attributes. Finding a satisfactory item from the repository may require several iterations of approximate search followed by browsing. With each iteration, the developer would get a better understanding of the available products and their differences. However, we must remember that the items to be searched may be components, designs, models, requirements, and even knowledge.

14.4.4 Repository Maintenance

Repository maintenance involves entering new items, retiring those items which are no more necessary, and modifying the search attributes of items to improve the effectiveness of search. Also, the links relating the different items may need to be modified to improve the effectiveness of search. The software industry is always trying to implement something that has not been quite done before. As patterns requirements emerge, new reusable components are identified, which may ultimately become more or less the standards. However, as technology advances, some components which are still reusable, do not fully address the current requirements. On the other hand, restricting reuse to highly mature components,can sacrifice potential reuse opportunity. Making a product available before it has been thoroughly assessed can be counter productive. Negative experiences tend to dissolve the trust in the entire reuse framework.

14.4.5 Reuse without Modifications

.Once standard solutions emerge, no modifications to the program parts

may be necessary. One can directly plug in the parts to develop his application. Reuse without modification is much more useful than the classical program libraries. These can be supported by compilers through linkage to run-time support routines (application generators).

Application generators translate specifications into application programs. The specification usually is written using 4GL. The specification might also be in a visual form. The programmer would create a graphical drawing using some standard available symbols. Defining what is variant and what is invariant corresponds to parameterising a subroutine to make it reusable. A subroutine's parameters are variants because the programmer can specify them while calling the subroutine. Parts of a subroutine that are not parameterised, cannot be changed.

Application generators have significant advantages over simple parameterised programs. The biggest of these is that the application generators can express the variant information in an appropriate language rather than being restricted to function parameters, named constants, or tables. The other advantages include fewer errors, easier to maintain, substantially reduced development effort, and the fact that one need not bother about the implementation details. Application generators are handicapped when it is necessary to support some new concepts or features. Some application generators overcome this handicap through an escape mechanism. Programmers can write code in some 3GL through this mechanism.

Application generators have been applied successfully to data processing application, user interface, and compiler development. Application generators are less successful with the development of applications with close interaction with hardware such as real-time systems.

14.5 REUSE AT ORGANISATION LEVEL

Reusability should be a standard part in all software development activities including specification, design, implementation, test, etc. Ideally, there should be a steady flow of reusable components. In practice, however, things are not so simple.

Extracting reusable components from projects that were completed in the past presents an important difficulty not encountered while extracting a reusable component from an ongoing project—typically, the original developers are no longer available for consultation. Development of new systems leads to an assortment of products, since reusability ranges from

items whose reusability is immediate to those items whose reusability is highly improbable.

Achieving organisation-level reuse requires adoption of the following steps:

- Assess of an item's potential for reuse.
- Refine the item for greater reusability.
- Enter the product in the reuse repository.

In the following subsections, we elaborate these three steps required to achieve organisation- level reuse.

Assessing a product's potential for reuse

Assessment of a components reuse potential can be obtained from an analysis of a questionnaire circulated among the developers. The questionnaire can be devised to assess a component's reusability. The programmers working in similar application domain can be used to answer the questionnaire about the product's reusability. Depending on the answers given by the programmers, either the component be taken up for reuse as it is, it is modified and refined before it is entered into the reuse repository, or it is ignored. A sample questionnaire to assess a component's reusability is the following:

- Is the component's functionality required for implementation of systems in the future?
- How common is the component's function within its domain?
- Would there be a duplication of functions within the domain if the component is taken up?
- Is the component hardware dependent?
- Is the design of the component optimised enough?
- If the component is non-reusable, then can it be decomposed to yield some reusable components?
- Can we parametrise a non-reusable component so that it becomes reusable?

Refining products for greater reusability

For a product to be reusable, it must be relatively easy to adapt it to different contexts. Machine dependency must be abstracted out or localised using data encapsulation techniques. The following

refinements may be carried out:

Name generalisation: The names should be general, rather than being directly related to a specific application.

Operation generalisation: Operations should be added to make the component more general. Also, operations that are too specific to an application can be removed.

Exception generalisation: This involves checking each component to see which exceptions it might generate. For a general component, several types of exceptions might have to be handled.

Handling portability problems: Programs typically make some assumption regarding the representation of information in the underlying machine. These assumptions are in general not true for all machines. The programs also often need to call some operating system functionality and these calls may not be the same on all machines. Also, programs use some function libraries, which may not be available on all host machines. A portability solution to overcome these problems is shown in Figure 14.1. The portability solution suggests that rather than call the operating system and I/O procedures directly, abstract versions of these should be called by the application program. Also, all platform-related calls should be routed through the portability interface. One problem with this solution is the significant overhead incurred, which makes it inapplicable to many real-time systems and applications requiring very fast response.

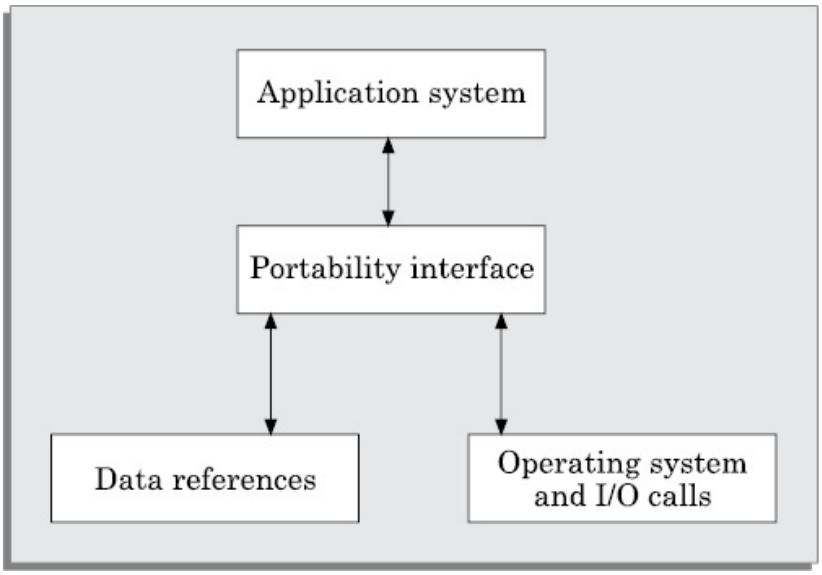

Figure 14.1: Improving reusability of a component by using a portability interface.

14.5.1 Current State of Reuse

In spite of all the shortcomings of the state-of-the-art reuse techniques, it is the experience of several organisations that most of the factors inhibiting an effective reuse program are non-technical. Some of these factors are the following:

- Need for commitment from the top management.
- Adequate documentation to support reuse.
- Adequate incentive to reward those who reuse. Both the people contributing new reusable components and those reusing the existing components should be rewarded to start a reuse program and keep it going.
- Providing access to and information about reusable components. Organisations are often hesitant to provide an open access to the reuse repository for the fear of the reuse components finding a way to their competitors.

SUMMARY

- We identified the following as the basic issues that must be addressed to initiate any meaningful reuse program—component creation; component indexing and storing; component search; component understanding; component adaptation; and repository maintenance
- Creation of highly reusable components is a very difficult problem. A promising approach is domain analysis. Domain analysis aims to identify reusable components for a problem domain.
- Application generators translate problem specifications into application programs. Applications generators greatly facilitate reuse compared to other ways of reusing components.
- We discussed reuse at organisation level. For this, three important steps are required to be followed. These are—assess an item's potential for reuse; refine the item for greater reusability, and enter the product in the reuse repository.

EXERCISES

1. Enumerate the major technical and non-technical reasons that hinder software reuse. Can there be circumstances where software reuse cannot be recommended?

2. Identify the important points that the developer of a software package must observe to enhance the reusability of the package.
3. Why is it important for an organisation to undertake an effective reuse program? What are the important reuse artifacts that can be reused? Why is reuse of software components much more difficult than hardware components?
4. Identify the reasons why reuse of mathematical software is so successful. Also, identify the reasons why the reuse of software components other than those of the mathematical software is difficult.
5. What do you understand by the term reuse domain? Explain domain analysis and how domain analysis leads to increased component reusability.
6. Do you agree with the statement: "code" is the most important artifact that can be reused during software development? Justify your answer.
7. What do you understand by the term domain analysis in the context of software design? What artifacts are produced after domain analysis? How does domain analysis increase software reusability?
8. Compare the advantages and disadvantages of a reuse program based on component library and another based on an application generator.
9. Explain how components can be created effectively for reuse.
10. Explain the important aspects (steps) in starting and maintaining an effective reuse program in a software development organisation.
11. How can you enhance the reusability of a function that you are writing?
12. Identify the stages through which a reuse domain progresses.
13. Explain why reuse is difficult in software development compared to hardware development.
14. Explain why reuse of mathematical functions is easier compared to reuse of non-mathematical functions.
15. What do you understand by the term "faceted classification" in the context of software reuse? How does faceted classification simplify component search in a component store?
16. Explain how the faceted component classification (Prieto-Diaz's scheme) can be used for approximate searching. What are the advantages of approximate searching over exact searching?
17. Suppose your team has developed a software product. How would you assess the potential reusability of the developed functions?

18. In an organisation level software reuse, identify aspects of a developed function which hamper its reusability. How can you improve reusability of the components you have identified for reuse?
19. What is an application generator? Why reuse is easier while using an application generator compared to a component library? What are the shortcomings of an application generator?
20. Devise a scheme to store software reuse artifacts. Explain how components can be searched in your scheme.

Chapter 15

EMERGING TRENDS

We had discussed in Chapter 1 that software engineering techniques have in the past evolved in response to the challenges posed to program development by the changing environment in which the programs run and also the changes to the types of applications required by the users. By changes to the environment, we mean the changes that occur to the different technologies that underlie computer hardware, system software, networking, and peripheral devices. Let us examine the way the environment has changed of late. This can indicate the challenges being posed to the software development principles. This in turn would give us some insight into the way in which the software engineering techniques are evolving of late.

The important changes to the environment that have occurred in the last two decades include the following:

- The prices of computers have dropped drastically in this period. At the same time, they have become more powerful. Now they can perform computations much faster and store much larger volumes of data. The sizes of computers have shrunk and laptops and palmtops are becoming popular.
- The Internet has become extremely popular. Internet connects millions of computers world-wide and makes enormous available to the users.
- Networking techniques have made rapid progress. The speed of data transfer has increased unbelievably and at the same time, the cost of networking computers has dropped dramatically. Just to give an example of currently supported speed of data transfer, desktops now come with a default 1Gbps network port.
- Mobile phones have dramatically captured imagination of all. The level of acceptance that mobile phones have achieved in less than a decade

appears like a chapter straight out of a science fiction book. Mobile phones are rapidly transforming themselves into handheld computing devices. In addition to high speed fixed line connections, GPRS and wireless LANs have become common place.

- Over the last decade, cloud computing has become popular. In cloud computing, applications are hosted on cloud operating on a data center. Cloud computing is becoming more and more popular as it helps a user run sophisticated applications without much upfront investments and also frees him from buying and maintaining sophisticated hardware and software.

In the face of the discussed developments, software developers are facing several challenges. Following are some of the challenges that are being faced by software developers.

Challenges faced by software developers

Following are some of the challenges that are being faced by software developers:

- To cope up with fierce competitions, business houses are rapidly changing their business processes. This requires rapid changes to also occur to the software that support the business process activities. Therefore, there is a pressing demand to shorten the software delivery time. However, software is still taking unacceptably long time to develop and is turning out to be a bottleneck in implementing rapid business process changes. To reduce the software delivery times, software is being developed by teams working from globally distributed locations. How software can be effectively developed using globally distributed development teams is not yet clear and poses many challenges. On the other hand, radical changes to the software development principles are being put forward to shorten the development time.
- Business houses are getting tired of astronomical software costs, late deliveries, and poor quality products. On the other hand, hardware costs are dropping and at the same time hardware is becoming more powerful, sophisticated, and reliable. Hardware and software cost differentials are becoming more and more glaring. The wisdom of developing every software from scratch is being questioned.Also,

alternate software delivery models are being proposed to reduce the software cost.
- Software sizes are further increasing.
- After Internet has become vastly popular, many software products are now required to interface with the Internet. Many products are even expected to work across the Internet. Also, with the availability of fast networks, distributed applications are becoming common place. However,it is not clear that how software is to be effectively developed in the context of distributed platforms and Internet.

In response to the challenges faced, the following software engineering trends are becoming noticeable:

- Client-server software
- Service-oriented architecture (SOA)
- Software as a service (SaaS)

In the following sections, we elaborate these emerging trends in software engineering.

15.1 CLIENT-SERVER SOFTWARE

In a client-server software, both clients and servers are essentially software components. A client is a consumer of services and a server is a provider of services. The client-server concept is not a new concept. It existed in the society since long. For example, a teacher may be a client of a doctor, and the doctor may in turn be a client of a barber, who in turn may be a client of the lawyer, and so forth. From this, we can observe that a server in some context can be a client in some other context. So, clients and servers can be considered to be mere roles. Considering the level of popularity of the client-server paradigm in the context of software development, there must be several advantages accruing from adopting this concept. Let us deliberate on the important advantages of the client-server paradigm.

Advantages of client-server software

There are many reasons for the popularity of client-server software. A few important reasons are as follows:

Concurrency: A client-server software divides the computing work among

many different client and server components that could be residing on different machines. Thus client-server solutions are inherently concurrent and as a result offer the advantage of faster processing.

Loose coupling: Client and server components are inherently loosely-coupled, making these easy to understand and develop.

Flexibility: A client-server software is flexible in the sense that clients and servers can be attached and removed as and when required. Also, clients can access the servers from anywhere.

Cost-effectiveness: The client-server paradigm usually leads to cost-effective solutions. Clients usually run on cheap desktop computers, whereas severs may run on sophisticated and expensive computers. Even to use a sophisticated software, one needs to own only a cheap client machine to invoke the server.

Heterogeneous hardware: In a client-server solution, it is easy to have specialised servers that can efficiently solve specific problems. It is possible to efficiently integrate heterogeneous computing platforms to support the requirements of different types of server software.

Fault-tolerance: Client-server solutions are usually fault-tolerant. It is possible to have many servers providing the same service. If one server becomes unavailable, then client requests can be directed to any other working server.

Mobile computing: Mobile computing implicitly requires uses of client-server technique. Cell phones are, of late, evolving as handheld computing and communicating devices and are being provided with small processing power, keyboard, small memory, and LCD display. The handhelds have limited processing power and storage capacity, and therefore can act only as clients. To perform any non-trivial task, the handheld computers can possibly only support the necessary user interface to place requests on some remote servers.

Application service provisioning: There are many application software products that are extremely expensive to own. A client-server based approach can be used to make these software products affordable for use. In this approach, an application service provider (ASP) would own it, and the users would pay the ASP based on the charges per unit time of usage.

Component-based development: Client-server paradigm fits well with the component- based software development. Component-based software

development holds out the promise of achieving substantial reductions to cost and delivery time and at the same time achieve increased product reliability. Component-based development is similar to the way hardware equipments are being constructed cost-effectively. A hardware developer achieves cost, effort, and time savings in an equipment development by integrating pre-built components (ICs) purchased off-the-shelf on a printed circuit board (PCB).

In the component paradigm, software development consists of integrating off-the-shelf software components and writing only the missing parts.

As discussed, advantages of the client-server software paradigm are numerous. No wonder that the client-server paradigm has become extremely popular. However, before we discuss more details of this technology, it is important to know the important shortcomings of it as well.

Disadvantages of client-server software

There are several disadvantages of client-server software development. The main disadvantages are:

Security: In a monolithic application, addressing the security concerns is much easier as compared to client-server implementations. A client-server based software provides many flexibilities. For example, a client can connect to a server from anywhere. This makes it easy for hackers to break into the system. Therefore, ensuring security of a client-server system is a very challenging task.

Servers can be bottlenecks: Servers can turn out to be bottlenecks because many clients might try to connect to a server at the same time. This problem arises due to the flexibility given that any client can connect anytime required.

Compatibility: Clients and servers may not be compatible to each other. Since the client and server components may be manufactured by different vendors, they may not be compatible with respect to data types, languages, number representation, etc.

Inconsistency: Replication of servers can potentially create problems as whenever there is replication of data, there is a danger of the data becoming inconsistent.

15.2 CLIENT-SERVER ARCHITECTURES

The simplest way to connect clients and servers is by using a two-tier

architecture shown in Figure 15.1(a). In a two-tier architecture, any client can get service from any server by sending a request over the network.

Limitations of two-tier client-server architecture

A two-tier architecture for client-server applications though is an intuitively obvious solution, but it turns out to be not practically usable. The main problem is that client and server components are usually manufactured by different vendors, who may adopt their own interfacing and implementation solutions. As a result, the different components may not interface with (talk to) each other easily.

Three-tier client-server architecture

The three-tier architecture overcomes the main limitations of the two-tier architecture. In the three-tier architecture, a middleware is added between client and the server components as shown in Figure 15.1(b). The middleware keeps track of all servers. It also translates client requests into server understandable form. For example, the client can deliver its request to the middleware and disengage because the middleware will access the data and return the answer to the client.

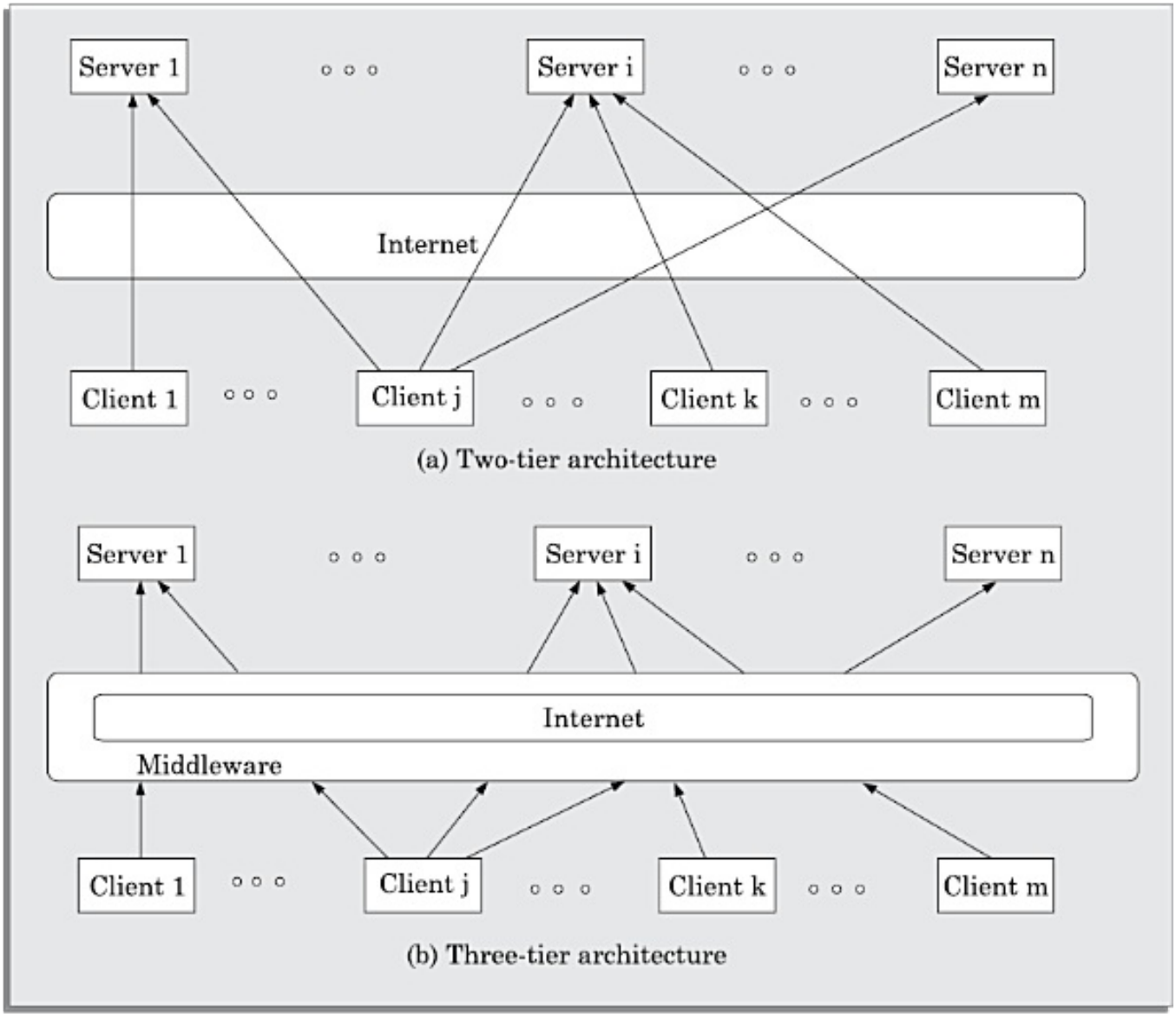

Figure 15.1: Two-tier and three-tier client-server architectures.

Functions of middleware

The important activities of the middleware include the following:

- The middleware keeps track of the addresses of servers. Based on a client request, it can therefore easily locate the required server.
- It can translate between client and server formats of data and vice versa.

Two popular middleware standards are:

- Common Object Request Broker Architecture (CORBA)
- COM/DCOM

CORBA is being promoted by Object Management Group (OMG), a consortium of a large number of computer industries such as IBM, HP, Digital, etc. However, OMG is not a standards body. OMG in fact does not have any authority to make or enforce standards. It just tries to popularize good solutions with the hope that if a solution becomes highly popular, it would ultimately become a standard. COM/DCOM is being promoted mainly by Microsoft. In the following subsections, we discuss these two important middleware standards.

15.3 CORBA

Common object request broker architecture (CORBA) is a specification of a standard architecture for middleware. Using a CORBA implementation, a client can transparently invoke a service of a server object, which can be on the same machine or across a network. CORBA automates many common network programming tasks such as object registration, location, and activation; request demultiplexing; framing and error-handling; parameter marshalling and demarshalling; and operation dispatching.

15.3.1 CORBA Reference Model

The CORBA reference model has been shown in Figure 15.2. In the following subsection, we briefly discuss the major components of the CORBA reference model.

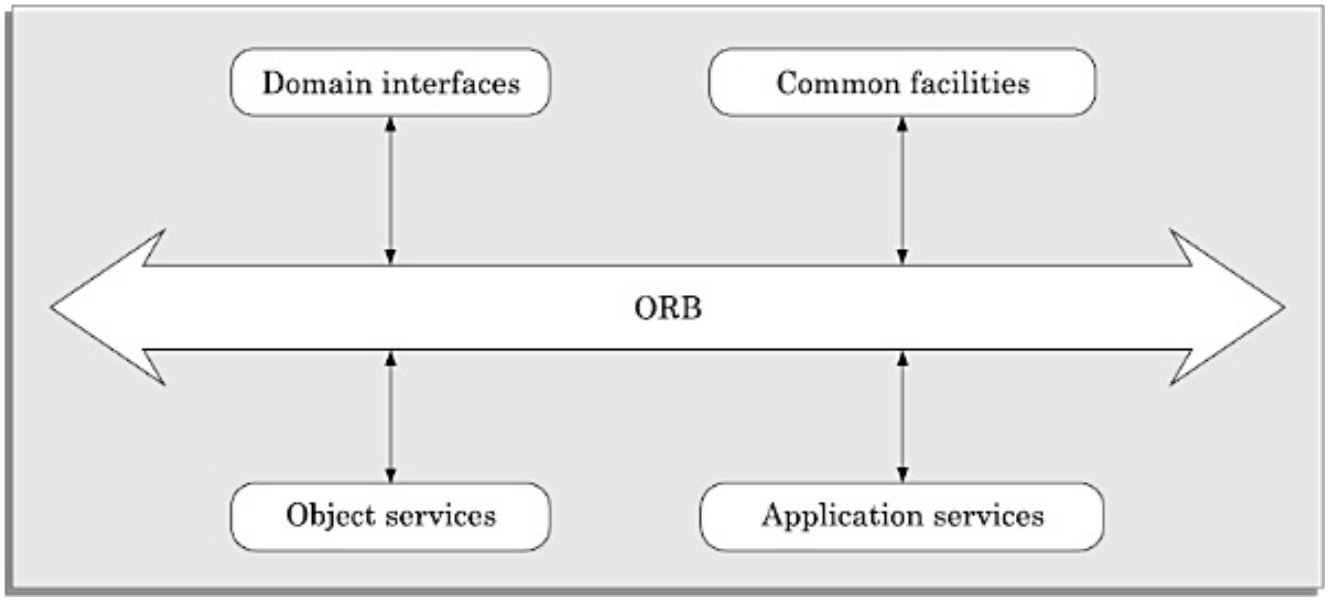

Figure 15.2: CORBA reference model.

ORB

ORB is also known as the **object bus**, since ORB supports communication among the different components attached to it. This is akin to a bus on a printed circuit board (PCB) on which the different hardware components (ICs) communicate. Observe that due to this analogy, even the symbol of a bus from the hardware domain is used to represent ORB (see Figure 15.2). The ORB handles client requests for any service, and is responsible for finding an object that can implement the request, passing it the parameters, invoking its method, and returning the results of the invocation. The client does not have to be aware of where the required server object is located, its programming language, its operating system or any other aspects that are not part of an object's interface.

Domain interfaces

These interfaces provide services pertaining to specific application domains. Several domain services have been in use, including manufacturing, telecommunication, medical, and financial domains.

Object services

These are domain-independent interfaces that are used by many distributed object programs. For example, a service providing for the discovery of other available services is almost always necessary regardless of the application domain. Two examples of object services that fulfill this role are the following:

Naming Service: This allows clients to find objects based on names. Naming service is also called white page service.

Trading Service: This allows clients to find objects based on their properties. Trading service is also called yellow page service. Using trading service a specific service can be searched. This is akin to searching a service such as automobile repair shop in a yellow page directory.

There can be other services which can be provided by object services such as security services, life-cycle services and so on.

Common facilities

Like object service interfaces, these interfaces are also horizontally-oriented, but unlike object services they are oriented towards end-user

applications. An example of such a facility is the distributed document component facility (DDCF), a compound document common facility based on OpenDoc. DDCF allows for the presentation and interchange of objects based on a document model, for example, facilitating the linking of a spreadsheet object into a report document.

Application interfaces

These are interfaces developed specifically for a given application.

15.3.2 CORBA ORB Architecture

The representation of Figure 15.3 is simplified since it does not show the various components of ORB. Let us now discuss the important components of CORBA architecture and how they operate. The ORB must support a large number of functions in order to operate consistently and effectively. In the carefully thought-out design of ORB, the ORB implements much of these functionality as pluggable modules to simplify the design and implementation of ORB and to make it efficient.

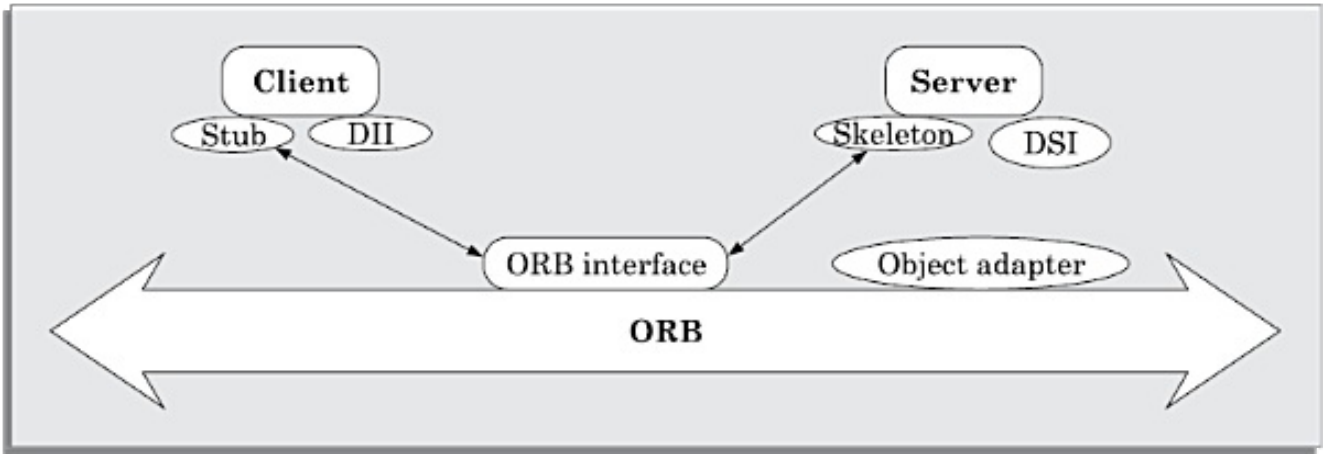

Figure 15.3: CORBA ORB architecture.

ORB

CORBA's most fundamental component is the object request broker (ORB) whose task is to facilitate communication between objects. The main responsibility of ORB is to transmit the client request to the server and get the response back to the client. ORB abstracts out the complexities of service invocation across a network and makes service invocation by client seamless and easy. The ORB simplifies distributed programming by decoupling clients from the details of the service

invocations. This makes client requests appear to be local procedure calls. When a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller. ORB allows objects to hide their implementation details from clients. The different aspects of a program that are hidden (abstracted out) from the client include programming language, operating system, host hardware,and object location.

Stubs and skeletons

Using a CORBA implementation clients can communicate to the server in two ways—by using stubs or by using dynamic invocation interface (DII). The stubs help static service invocation, where a client requests for a specific service using the required parameters. In the dynamic service invocation, the client need not know before hand about the required parameters and these are determined at the run time. Though dynamic service invocation is more flexible, static service invocation is more efficient that dynamic service invocation.

Service invocation by client through stub is suitable when the interface between the client and server is fixed and it does not change with time. If the interface is known before starting to develop client and the server parts then stubs can effectively be used for service invocation. The stub part resides in the client computer and acts as a proxy for the server which may reside in the remote computer. That is the reason why stub is also known as a proxy.

Object adapter

Service invocation through dynamic invocation interface (DII) transparently accesses the interface repository (OA). When an object gets created, it registers information about itself with OA. DII gets the relevant information from the IR and lets the client know about the interface being used.

15.3.3 CORBA Implementations

There are several CORBA implementations that are available for use. The following are a few popular ones.

- Visibroker is a software from Borland is probably the most popular CORBA implementation. Netscape browser supports Visibroker.

Therefore, CORBA applications can be run using Netscape web browser. In other words, Netscape browser can act as a client for CORBA applications. Netscape is extremely popular and there are several millions of copies installed on desktops across the world.

- Orbix from Iona technologies.
- Java IDL.

15.3.4 Software Development in CORBA

Let us examine how software can be developed in CORBA. Before developing a client-server application, the solution is split into two parts —the client part and the serv part. Next, the exact client and server interfaces are determined. To specify an interface, interface definition language (IDL) is used. IDL is very similar to C++ and Java except that it has no executable statements. Using IDL only data interface between clients and servers can be defined. It supports inheritance so that interfaces can be reused in the same or across different applications. It also supports exception.

After the client-server interface is specified in IDL, an IDL compiler is used to compile the IDL specification. Depending on whether the target language in which the application is to be developed is Java, C++, C, etc., Different IDL compilers such as IDL2Java, IDL2C++, IDL2C etc. can be used as required. When the IDL specification is compiled, it generates the skeletal code for stub and skeleton. The stub and skeleton contain interface definitions and only the method body needs to be written by the programmers developing the components.

Inter-ORB communication

Initially, CORBA could only integrate components running on the same LAN. However, on certain applications, it becomes necessary to run the different components of the application in different networks. This shortcoming of CORBA 1.X was removed by CORBA 2.0. CORBA 2.0 defines general interoperability standard. The general inter-orb protocol (GIOP) is an abstract meta-protocol. It specifies a standard transfer syntax and a set of message formats for object requests. The GIOP is designed to work over many different transport protocols. In a distributed implementation, every ORB must support GIOP mapped onto its local transport. GIOP can be used by almost any connection-oriented byte stream transport.

GIOP is popularly implemented on TCP/IP known as internet inter-ORB protocol (IIOP).

15.4 COM/DCOM

15.4.1 COM

The main idea in the component object model (COM) is that different vendors can sell binary components. Application can be developed by integrating off-the-shelf components. COM can be used to develop component applications on a single computer. The concepts used are very similar to CORBA. The components are known as binary objects. These can be generated using languages such as Visual Basic, Delphi, Visual C++ etc. These languages have the necessary features to create COM components. COM components are binary objects and they exist in the form of either .exe or .dll (dynamic link library). The .exe components have separate existence. But .dll COM components are in-process servers, that get linked to a process. For example, ActiveX is a dll type server, which gets loaded on the client-side.

15.4.2 DCOM

Distributed component object model (DCOM) is the extension of the component object model (COM). The restriction that clients and servers reside in the same computer is relaxed here. So, DCOM can operate on networked computers. Using DCOM, development is easy as compared to CORBA. Much of the complexities are hidden from the programmer.

15.5 SERVICE-ORIENTED ARCHITECTURE (SOA)

Service-orientation principles have their roots in the object-oriented designing. Many claim that service-orientation will replace object-orientation; others think that the two are complementary paradigms.

SOA views software as providing a set of services. Each service composed of smaller services. Let us first understand what are software services. Services are implemented and provided by a component for use by an application developer. A service is a contractually defined behaviour. That is, a component providing a service guarantees that its behaviour is as per the specifications. A few examples of services are the following—Filling out an on-line application, viewing an on-line bank-statement, and placing an online booking. Different services in an application communicate with each other.

The services are self-contained. That is, a service does not depend on the context or state of the other service. An application integrating different services works within a distributed-system architecture.

The main idea behind SOA is to build applications by composing software services.

SOA principally leverages the Internet and emerging the standardisations on it for interoperability among various services. An application is built using the services available on the Internet, and writing only the missing ones.

There are several similarities between services and components, which are as follows:

- **Reuse:** Both a component and a service are reused across multiple applications.
- **Generic:** The components and services are usually generic enough to be useful to a wide range of applications.
- **Composable:** Both services and components are integrated together to develop an application.
- **Encapsulated:** Both components and services are non-investigable through their interfaces.
- **Independent development and versioning:** Both components and services are developed independently by different vendors and also continue to evolve independently.
- **Loose coupling:** Both applications developed using the component paradigm and the SOA paradigm have loose coupling inherent to them.

However, there are several dissimilarities between the components and the SOA paradigm, which are as follow:

- The granularity (size) of services in the SOA paradigm are often 100 to 1,000 times larger than the components of the component paradigm.
- Services may be developed and hosted on separate machines.
- Normally components in the component paradigm are procured for use as per requirement (ownership). On the other hand, services are usually availed in a pay per use arrangement.

Instead of services embedding calls to each other in their source code, services use well-defined protocols which describe how services can talk to each other. This architecture facilitates a business process expert to tailor an application as per requirement. To meet a new business requirement, the

business process expert can link and sequences services in a process known as orchestration.

SOA targets fairly large chunks of functionality to be strung together to form new services. That is, large services can be developed by integrating existing software services. The larger the chunks, the fewer the interfacings required. This leads to faster development. However, very large chunks may prove to be difficult to reuse.

15.5.1 Service-oriented Architecture (SOA): Nitty Gritty

The SOA paradigm utilises services that may be hosted on different computers. The different computers and services may be under the control of different owners. To facilitate application development, SOA must provide a means to offer, discover, interact with and use capabilities of the services to achieve desired results.

SOA involves statically and dynamically plugging-in services to build software. SOA players—BEA Aqua logic, Oracle Web services manager, HP Systinet Registry, MS .Net, IBM Web Sphere, Iona Artrix, Java composite application suite. Web services can be used to implement a service-oriented architecture. Web services can make functional building blocks accessible over standard Internet protocols independent of platforms and programming languages.

One of the central assumptions of SOA is that once a market place for services develops, services can be purchased to develop new applications. To build an application, one would use off-the-shelf services and possibly build some. When services are used across a large number of applications, automatically quality would improve and also price would reduce. When a service is used by a very large number of applications, the cost of using that service becomes near zero. Thus the cost of creating an application that uses widely used services would also be near zero, as all of the software services required would already exist and cost near zero, only orchestration of these services would be required to produce the application.

15.6 SOFTWARE AS A SERVICE (SAAS)

Owning software is very expensive. For example, a Rs. 50 Lakh software running on an Rs. 1 Lakh computer is common place. As with hardware, owning software is the current tradition across individuals and business houses. Most of IT budget now goes in supporting the software assets. The support cost includes annual maintenance charge (AMC), keeping

the software secure and virus free, and taking regular back-ups, etc. But, often the usage of a specific software package does not exceed a couple of hours of usage per week. In this situation, it would be economically worthwhile to pay per hour of usage. This would also free the user from the botherance of maintenance, upgradation, backup, etc. This is exactly what is advocated by SaaS.

In this context, SaaS makes a case for pay per usage of software rather than owning software for use.

SaaS is a software delivery model and involves customers to pay for any software per unit time of usage, with the price reflecting market place supply and demand.

As we can see, SaaS shifts "ownership" of the software from the customer to a service provider. Software owner provides maintenance, daily technical operation, and support for the software. Services are provided to the clients on amount of usage basis. The service provider is a vendor who hosts the software and lets the users execute on-demand charges per usage units. It also shifts the responsibility for hardware and software management from the customer to the provider. The cost of providing software services reduces as more and more customers subscribe to the service. Elements of outsourcing and application service provisioning are implicit in the SaaS model.Also, it makes the software accessible to a large number of customers who cannot afford to purchase the software outright. Target the "long tail" of small customers.

If we compare SaaS to SOA, we can observe that SaaS is a software delivery model, whereas SOA is a software construction model. Despite significant differences, both SOA and SaaS espouse closely related architecture models. SaaS and SOA complement each other. SaaS helps to offer components for SOA to use. SOA helps to help quickly realise SaaS. Also, the main enabler of SaaS and SOA are the Internet and web services technologies.

SUMMARY

- Some of the basic assumptions of software are changing. This is leading to some different paradigms for software development and delivery.
- Component-based development is expected to reduce development time, cost and at the same time improve quality.

- SaaS is changing the way software is delivered.
- SOA would fundamentally change the way we construct software systems. In the SOA paradigm, an application can be built by orchestrating existing services, and writing only the missing ones.

EXERCISES

1. Choos the correct option:
   (a) What are the possible reasons behind the recent popularity of the client-server style of software development?
   (i) Computers have become small, decentralised and cheap
   (ii) Networking has become affordable, reliable, and efficient
   (iii) Client-server systems divide up the work of computing among many separate machines
   (iv) All of the above
   (b) Which of the following functions are performed by middleware?
   (i) Identifying a server from either its id or its service type
   (ii) Converting between client protocols and server protocols
   (iii) Delivering client-request to the server and server-response to the client
   (iv) All of the above
   (c) Which of the following functions does object request broker (ORB) perform?
   (i) Transmit the client request to the server and get the response back to the client
   (ii) Location and possible activation of remote objects
   (iii) Interface definition
   (iv) All of the above
   (d) Before developing client-server application in CORBA, interface between the client part and the server must be specified using:
   (i) Interface definition language
   (ii) Dynamic invocation interface
   (iii) ORB
   (iv) None of the above
   (e) In CORBA if the server interface known to be invariant with time, then it is more efficient to use
   (i) Stubs and skeletons
   (ii) DSI and DII
   (iii) RMI

(iv) None of the above

2. What are the advantages of client-server software as compared to monolithic software? Also, identify the disadvantages of the client-server software.
3. What is common object request broker architecture (CORBA)? Explain CORBA architecture.
4. Identify three functions of object request broker (ORB).
5. Name at least three commercial ORBs.
6. Explain what is stub.
7. Explain dynamic invocation interface (DII) in CORBA.
8. Explain what are component object model (COM) and distributed component object model (DCOM).
9. Explain Inter-ORB communication.
10. What are the advantages of client-server software development?
11. Why is a two-tier architecture not a practical client-server architecture? How does the three tier architecture overcome the problems of the two-tier architecture.
12. What are the functions of a middleware in a three-tier architecture? Mention two popular middleware standards.
13. How does dynamic invocation interface (DII) know what format data or what exact data required by the server for providing the service and how does the client recognise the data?
14. What is general inter-ORB protocol (GIOP)? What are the features of GIOP?
15. Mark the following statements as either True or False. Justify your answer.
    (a) Fault-tolerance is more difficult to provide in a monolithic application compared to its client-server implementation.
    (b) Client-server based software development is more secure than a monolithic software.
    (c) Two-tier client-server architecture can be used to effectively develop commercial component-based systems.
    (d) The object adapter component in CORBA is responsible for translating the client data formats into server data formats and vice versa.
    (e) CORBA is the name of a software product that facilitates development of client- server solutions.
    (f) A CORBA-based client-server solution is constrained to run on a

single local area network (LAN).

(g) Using internet inter-ORB protocol (IIOP), a web browser such as Netscape can serve as a CORBA client.

16. Differentiate between monolithic and client-server software solutions. Give an example for each.

REFERENCES

Albrecht, A.J. and J.E. Gaffney, "Software function, lines of code, and development effort prediction: a software science validation," IEEE Trans. on Software Engineering, **9**(6), pp. 639–647.

Alhir, Sinan Si., UML in Nutshell, OReily, 1998.

Boehm, B.W., Software Engineering Economics, Prentice Hall, Englewood Cliffs, New Jersey, 1981.

Boehm, Barry, B. Clark and E. Horowitz, et al. "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0," Annals of Software Engineering, 1995.

Booch, Grady, Object-Oriented Design with Applications, Benjamin Cummings, Menlo Park, California, 1991.

Booch, Rumbaugh, Jacobson, The UML User's Guide, Addison-Wesley, Reading, Mass., 1999.

Brooks F., The Mythical Man-Month, Addison-Wesley, Reading, Mass., 1975.

Constantine, L.L. and E. Yourdon, Structured Design, Prentice Hall, Englewood Cliffs, New Jersey, 1979.

Conte, S.D., H.E. Dunsmore and V.Y. Shen, Software Engineering Metrics and Models, Benjamin/Cummings Publishing Company Inc., Menlo Park, California, 1986.

Corrado Bohm and Giuseppe Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", Communications of the ACM, **9**(5), May 1966.

Crosby, Philip B., Quality is Free, McGraw-Hill, New York, 1979.

DeMarco, T., Structured Analysis and System Specification, Yourdon Press, New York, 1978.

Dijkstra, E.W., "Goto Statement Considered Harmful," Communications of the ACM, **11**(3), pp. 147–148, 1968.

Eliason, Alan L., Systems Development Analysis and Implementation, Little

Brown and Company (Canada) Ltd., 1987.

Fairley, Richard, Managing and Leading Software Project, IEEE Press, 2009.

Fowler, Martin, UML Distilled, 2nd ed., Addison-Wesley, Reading Mass., 2000.

Gane, C. and T. Sarson, Structured Systems Analysis, Prentice Hall, Englewood Cliffs,
New Jersey, 1979.

George A., Miller, "The Magical Number Seven Plus or Minus Two: Some Limits on Our Capacity for Processing Information," The Psychological Review, **63**(2), pp. 8197, 1956.

Ghezzi, Carlo, Mehdi Jazayeri and Dino Mandrioli, Fundamentals of Software Engineering, Pearson Education Inc., 2003.

Guttag, J., "Notes on Type Abstraction," IEEE Transactions on Software Engineering, **6**(1), pp. 13–23, 1980.

Guttag, J., J. Horning, J.M. Wing, "The Larch Family of Specification Languages," IEEE Software, **2**(5), pp. 24–36, 1985.

Harel, D. et al., "Statemate: A working environment for the development of complex reactive systems," IEEE Transactions on Software Engineering, **16**(3), 403–414,
April 1990.

Hatley, D. and I. Pirbhai, Strategies for Real-Time System Specifications, Dorset House,
New York, 1987.

Hoare, C.A.R., "Programming Sorcery or Science?" IEEE Transactions on Software Engineering, pp. 5–16, April 1994.

Humphrey, Watts S., Introduction to the Personal Software Process, Addison Wesley, Longman, 1997.

IEEE Recommended Practice for Software Requirements Specifications, IEEE Std. 830–1998.

IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 610.12–1990.

Ince, Darrel, ISO 9001 and Software Quality Assurance, McGraw-Hill, New York, 1994.

Ince, David (Ed.), Software Quality and Reliability, Chapman and Hall Publishing, London, 1991.

Jackson, M.A., Principles of Program Design, Academic Press Inc., Orlando, Florida, 1975.

Jacobson, Booch, Rumbaugh, The Unified Software Development Process, Addison-Wesley, Reading Mass., 1999.

Jacobson, I. and Christerson, M., Object-oriented Software Engineering—A Use Case-Driven Approach, Addison-Wesley, England, 1992.

Jalote, Pankaj, An Integrated Approach to Software Engineering, Third Edition,
Springer, 2005.

Jelinski, Z. and P. Moranda, "Software Reliability Research," in Freiberger W. (Ed.), Statistical Computer Reliability Evaluation, Academic Press, pp. 465–484, 1972.

Jensen, R.W., "A Comparison of the Jensen and COCOMO Schedule and Cost Estimation Models," in Proc. of International Society of Parametric Analysis, pp. 96–106, 1984.

Jones, C.B., Software Development: A rigorous approach, Prentice Hall, Englewood Cliffs, New Jersey, 1980.

Lamb, David Alex, Software Engineering Planning for Change, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

Larman, Craig, Applying UML and Patterns—An Introduction to Object-Oriented Analysis and Design, Pearson Press, 1998.

Lawrence Pfleeger, Shari, Software Engineering Theory and Practice, Fourth Edition, Prentice Hall, Englewood Cliffs, New Jersey, 2009.

Leathrum, J.F., Foundations of Software Design, Reston Publishing Company, Virginia, USA, 1983.

Lehman, M.M. and L.A. Belady, "Programs Life Cycles and Laws of Software Evolution," Proceedings of IEEE, pp. 1060–1076, September 1980.

Ludolph, Frank, Model-based user interface designSuccessive transformations of a task object model, in User interface designbridging the gap from user requirements to design, CRC Press, 1998.

Martin, James and Carma McClure, Structured Techniques: The Basis for CASE, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

Martin, James and James J. Odell, Principles of Object-Oriented Analysis and Design, Prentice Hall, Englewood Cliffs, New Jersey, 1992.

McCabe, T., "A Complexity Measure," IEEE Transactions on Software Engineering, **4**(2)
December 1976.

Miller, G.A., "The Magical Number Seven, Plus or Minus Two: Some

Limits on Our Capacity for Processing Information," in the Psychological Review, **63**(2), pp. 81-97, March 1956.

Mund, G.B., R. Mall and S. Sarkar, "An Efficient Dynamic Program Slicing Technique," Journal of Information and Software Technology, Elsevier Press, **44**(2), pp. 123–132, March 2002.

Nielson, J. and R.L. Mack, Usability Inspection Methods, John Wiley, 1994.

Orr, K., Structured Requirements Definition, Ken Orr and Associates, Australia, 1981.

Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, **15**(2), pp. 105–358, 1972.

Pressman, Roger S., Software Engineering, 7th Ed., McGraw-Hill, New York, 2009.

Putnam, L.H., "A General Empirical Solution to Macro Software Sizing and Estimation Problem," IEEE Transactions on Software Engineering, **4**(3), pp. 345–361, 1978.

Rosenberg D., Use Case-Driven Object Modelling with UML, Addison-Wesley, Reading, Mass., 2000.

Rumbaugh, J., I. Jacobson and G. Booch, The UML Reference Manual, Addison-Wesley, Reading, Mass., 1999.

Rumbaugh, J., M. Blaha, et al., Object-Oriented Modelling and Design, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Sackman H., W.J. Erikson and E.E. Grant "Exploratory Experimentation Studies Comparing On-line and Off-line Programming Performance," Communications of the ACM, **11**(1), pp. 3–11, 1968.

Scheifler, Robert W., Jim Gettys and R. Newman, X Window System—C Library and Protocol Reference—C Library and , DEC Press, Bedford, Mass., 1988.

Shlaer, Sally and Stephen J. Mellor, Object Lifecycles: Modeling the

World in States, Prentice Hall, Englewood Cliffs, New Jersey, 1992.

Smith, David J. and Kenneth B. Wood, Engineering Quality Software, Elsevier Science Publishing, New York, 1987.

Somerville, Ian, Software Engineering, 9th ed., Addison-Wesley, Reading, Mass., 1992.

Ward, P. and S. Mellor, Structured Development of Real-Time Systems, Prentice Hall, Englewood Cliffs, New Jersey, 1985.

Warnier, J.D., Logical Construction of Programs, Van Nostrand Reinhold, New York, 1977.

Wirfs-Brock, Rebecca, Brian Wilkerson, Lauren Wiener, Designing Object Oriented Software, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

Wirth, N., "Program Development by Stepwise Refinement," Communications of the ACM, **14**(4), pp. 221–227, 1971.

INDEX

Abstract class, 287
Abstraction, 15, 289
Acceptance testing, 436
Activity diagram, 320
Activity networks, 122
Adaptive maintenance, 44
Aggregation, 286, 314
Agile development models, 62
Algebraic specification, 188
Alpha testing, 436
Antipattern, 341
Architectural patterns, 339
Association, 284, 314
Axiomatic specification, 186
Basis path set, 422

Beta testing, 436
Black-box testing, 414
Booch's object identification method, 357
Boundary value analysis, 415
Branch coverage, 419

Capability maturity model integration, 477
Case, 485
Characteristics of software maintenance, 494
Class, 280
Class diagrams, 311
Class relationships, 281
Class-responsibility-collaborator (CRC)
Cards, 360
Classical waterfall model, 38
Clean room testing, 403
Client-server software, 514
CMM, 473
COCOMO, 101
COCOMO 2, 109
Code inspection, 401, 402
Code review, 400
Code walkthrough, 401
Coding, 398
Coding and unit testing, 43
Coding standards, 399
Cohesion, 208
Collaboration diagram, 319

COM/DCOM, 522
Command language-based interface, 378
Compatibility testing, 437
Complete COCOMO, 108
Component adaptation, 505
Component classification, 506
Component creation, 504
Component diagram, 324
Component indexing, 504
Component searching, 505
Component understanding, 505
Composition, 286, 315
Computer systems engineering, 27
Configuration testing, 437
Constraints, 317
Control flow-based design, 18
Control flow graph (CFG), 421
CORBA, 518
Corrective maintenance, 44
Coupling, 208, 211
Creator, 342
Critical path method, 124
Cyclomatic complexity, 423
Data flow diagram (DFD), 224, 225
Data flow-based testing, 425
Data flow-oriented design, 22
Data hiding, 290
Data structure-oriented design, 22
Debugging, 427
Decision table, 181
Decision tree, 180
Decomposition, 16
Delphi cost estimation, 101
Dependency, 287, 316
Deployment diagram, 324
Design, 42
Design patterns, 339
Design review, 253
Detailed design, 253
Direct manipulation interfaces, 378, 381
Documentation testing, 438
Domain modelling, 353
Driver, 413, 414
Dynamic analysis, 429
Dynamic binding, 291
Encapsulation, 290,
Equivalence class partitioning, 414
Equivalent faults, 408
Error seeding, 438
Estimation of maintenance cost, 500

Evolution of quality systems, 466
Evolutionary model, 57
Executable specification, 193
Expert, 341
Expert judgement, 100
External documentation, 404
Extreme programming model, 66
Failure mode, 408
Feasibility study, 40
Fog index, 405
Formal methods, 182
Function point, 94
Function-oriented design, 214
Functional independence, 208
Functional requirements, 165, 167
Gantt charts, 128
Genericity, 294
Goals of implementation, 165, 167
Grey-box testing of object-oriented programs, 434
Halstead's software science, 112
High-level design, 43
Idioms, 339
Incremental development model, 55
Inheritance, 281, 316
Integration and system testing, 43
Integration testing, 430
Interaction diagrams, 318
Interaction modelling, 360
Intermediate COCOMO, 107
Internal documentation, 404
ISO 9000, 467
Iterative waterfall model, 46
Layered design, 207, 212
Linearly independent set of paths, 422
Lines of Code (LOC), 92
Low-level design, 43
Maintenance, 44
Maintenance testing, 438
Menu-based interface, 378, 379
Method overriding, 292
Methods, 280
Model view separation patterns, 343
Model-view-controller (MVC) pattern, 346
Modularity, 206
MTTF, 460, 461
MTTR, 461
Multiple condition coverage, 420

Mutation testing, 426
Non-functional requirements, 165, 166

Object diagrams, 317
Object-orientation concepts, 277
Object-oriented design, 23, 215
Object-oriented design approach, 43
Objects, 277
Observer pattern, 345
Organisation structure, 129
Package diagram, 323
Path coverage, 421
Patterns, 337
Perceived problem complexity, 11
Perfective maintenance, 44
Performance testing, 436
Person-month, 103
Personal software process (PSP), 477
PERT charts, 126
Phase containment of errors, 47
Phase entry and exit criteria, 38
POFOD, 461
Polymorphism, 291
Procedural design approach, 42
Process versus methodology, 35
Process metrics, 467
Product metrics, 467
Program analysis, 428
Programs versus products, 6
Project estimation techniques, 99
Project planning, 89
Prototyping model, 52
Publish-subscribe pattern, 346
Rapid application development, 59
Recovery testing, 438
Regression testing, 438
Reliability growth modelling, 462
Reliability metrics, 460
Repository maintenance, 505, 507
Requirements analysis, 159
Requirements analysis and specification, 42
Requirements gathering, 156
Requirements gathering and analysis, 42
Requirements specification, 42
Risk assessment, 138
Risk identification, 137
Risk management, 136
Risk mitigation, 138
ROCOF, 460
Scheduling, 119

Scrum model, 69
Security testing, 438
SEI capability maturity model, 473
Sequence diagram, 318
Service-oriented architecture (SOA), 522
Six sigma, 479
Sliding window planning, 90
Smoke testing, 436
Software architecture, 43
Software as a Service (SaaS), 524
Software configuration management, 140
Software crisis, 5
Software development life cycle (SDLC) model, 34
Software documentation, 403
Software life cycle, 34
Software maintenance, 494
Software maintenance process models, 497
Software process improvement and capability determination (SPICE), 477
Software products, 7
Software quality, 464
Software quality management, 465
Software re-engineering, 498
Software reliability, 458
Software requirements specification, 161
Software reuse, 503
Software reverse engineering, 496
Software services, 7
Spiral model, 69
SPMP document, 90
Staff-size estimation, 107
Staffing, 135
Staffing level estimation, 116
State chart diagram, 322
Statement coverage, 419
Static analysis, 428
Statistical testing, 463
Stress testing, 437
Structure chart, 248
Structured analysis, 42, 225
Structured design, 42, 247
Stub, 413
System testing, 435
Team structure, 132
Test documentation, 439
Test suite, 408
Testability, 408
Testing, 405
Top-down decomposition, 214
Transaction analysis, 250
Transform analysis, 249
UML 2.0, 325

Unadjusted function points, 95
Unified Modelling Language (UML), 296
Unified process, 349
Unit testing, 413
Usability testing, 438
Use case, 351
Use case model, 302
User interface design, 373
V-Model, 50
Validation, 409
Verification, 409
Visual programming, 384
Volume testing, 437
White-box testing, 417
Widgets, 385
Window manager, 383
Window system, 382
Work breakdown structure, 121
X-Window, 386