Global Institute of Technology, Jaipur
ITS-1, IT Park, EPIP, Sitapura Jaipur 302022 (Rajasthan)
Department of Computer Science & Engineering

Subject: SOFTWARE ENGINEERING

III Semester Subject Code: 3CS4-07

SYLLABUS

UNIT 1: Introduction, software life-cycle models, software requirements specification, formal requirements specification, verification and validation.

UNIT 2: Software Project Management: Objectives, Resources and their estimation, LOC and FP estimation, effort estimation, COCOMO estimation model, risk analysis, software project scheduling.

UNIT 3: Requirement Analysis: Requirement analysis tasks, Analysis principles. Software prototyping and specification data dictionary, Finite State Machine (FSM) models. **Structured Analysis**: Data and control flow diagrams, control and process specification behavioral modeling

UNIT 4: Software Design: Design fundamentals, Effective modular design: Data architectural and procedural design, design documentation.

UNIT 5: Object Oriented Analysis: Object oriented Analysis Modeling, Data modeling. **Object Oriented Design**: OOD concepts, Class and object relationships, object modularization, Introduction to Unified Modeling Language

UNIT-I: INTRODUCTION TO SOFTWARE ENGINEERING

The term software engineering is composed of two words, software and engineering.

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product.**

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods. So, we can define software engineering as an engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures. **The outcome of software engineering is an efficient and reliable software product.**

IEEE defines software engineering as: The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.

We can alternatively view it as a systematic collection of past experience. The experience is arranged in the form of methodologies and guidelines. A small program can be written without using software engineering principles. But if one wants to develop a large software product, then software engineering principles are absolutely necessary to achieve a good quality software cost effectively.

NEED OF SOFTWARE ENGINEERING

The need of software engineering arises because of higher rate of change in user requirements and environment on which the software is working.

Large software - It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering has to step to give it a scientific process.

Scalability- If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.

Cost- As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.

Dynamic Nature- The always growing and adapting nature of software hugely depends upon the environment in which the user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.

Quality Management- Better process of software development provides better and quality software product.

CHARACTERESTICS OF GOOD SOFTWARE

A software product can be judged by what it offers and how well it can be used. This software must satisfy on the following grounds:

- **Operational**
- **Transitional**
- **Maintenance**

Well-engineered and crafted software is expected to have the following characteristics:

Operational: This tells us how well software works in operations. It can be measured on:

- Budget
- Usability
- Efficiency
- Correctness

- Functionality
- Dependability
- Security
- Safety

Transitional: This aspect is important when the software is moved from one platform to another:

- Portability
- Interoperability
- Reusability
- Adaptability

Maintenance: This aspect briefs about how well a software has the capabilities to maintain itself in the ever-changing environment:

- Modularity
- Maintainability
- Flexibility
- Scalability

In short, Software engineering is a branch of computer science, which uses well-defined engineering concepts required to produce efficient, durable, scalable, in-budget and on-time software products.

LAYERS OF SOFTWARE ENGINEERING

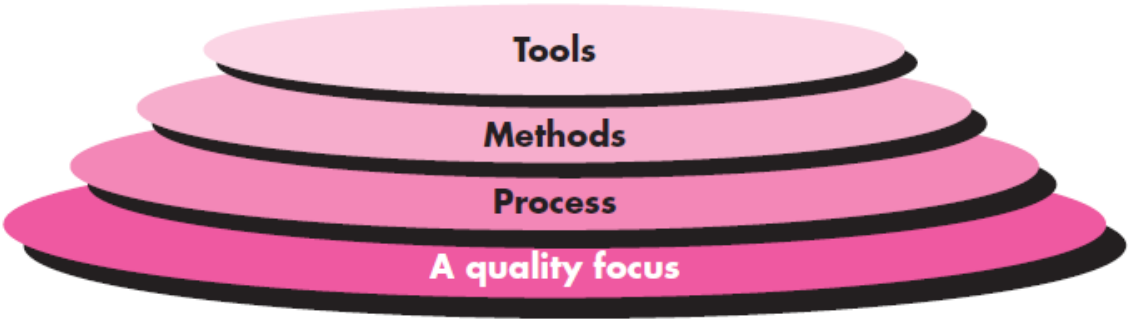

Figure: Layers Of Software Engineering

Software engineering is a layered technology. Software engineering must rest on an organizational commitment to **quality**. Total quality management, Six Sigma, and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is a **quality focus.**

A **Software engineering process** is *not* a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks. The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

Software engineering methods provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

Software engineering tools provide automated or semiautomated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established.

THE SOFTWARE PROCESS

A **process** is a collection of activities, actions, and tasks that are performed when some work product is to be created.

An **activity** strives to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied.

An **action** (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model).

A **task** focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

In the context of software engineering, a process is not a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks.

The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

A GENERIC PROCESS FRAMEWORK FOR SOFTWARE ENGINEERING: A **process framework** establishes the foundation for a complete software engineering process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity.

In addition, the process framework encompasses a set of umbrella activities that are applicable across the entire software process.

A generic process framework for software engineering encompasses five activities:

Communication: Before any technical work can commence, it is critically important to communicate and collaborate with the customer and other stakeholders and stakeholders' objectives for the project and to gather requirements that help define software features and functions.

Planning: Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a software project plan—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

Modeling: Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

Construction: This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

Deployment: The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These five generic framework activities can be used during the development of small, simple programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems. The details of the software process will be quite different in each case, but the framework activities remain the same.

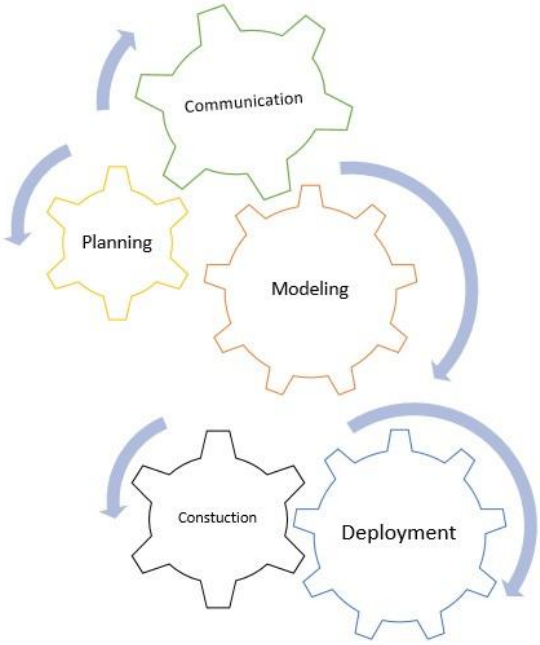

Figure: Activities in Generic process framework for software engineering

SOFTWARE LIFE CYCLE MODELS/SOFTWARE DEVELOPMENT LIFE CYCLE MODEL (SWDLC/SDLC MODELS)

A software life cycle model (also called process model) is a descriptive and diagrammatic representation of the software life cycle.

A life cycle model represents all the activities required to make a software product transit through its life cycle phases. It also captures the order in which these activities are to be undertaken. In other words, a life cycle model maps the different activities performed on a software product from its inception to retirement.

Different life cycle models may map the basic development activities to phases in different ways. Thus, no matter which life cycle model is followed, the basic activities are included in all life cycle models though the activities may be carried out in different orders in different

life cycle models. During any life cycle phase, more than one activity may also be carried out.

THE NEED FOR A SOFTWARE LIFE CYCLE MODEL

The development team must identify a suitable life cycle model for the particular project and then adhere to it. Without using of a particular life cycle model the development of a software product would not be in a systematic and disciplined manner.

When a software product is being developed by a team there must be a clear understanding among team members about when and what to do. Otherwise it would lead to chaos and project failure.

This problem can be illustrated by using an example. Suppose a software development problem is divided into several parts and the 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, another might decide to prepare the test documents first, and some other engineer might begin with the design phase of the parts assigned to him. This would be one of the perfect recipes for project failure.

A software life cycle model defines entry and exit criteria for every phase. A phase can start only if its phase-entry criteria have been satisfied. So without software life cycle model the entry and exit criteria for a phase cannot be recognized. Without software life cycle models it becomes difficult for software project managers to monitor the progress of the project.

Different software life cycle models

Many life cycle models have been proposed so far. Each of them has some advantages as well as some disadvantages. A few important and commonly used life cycle models are as follows:

Types of Software developing life cycles (SDLC)

- Waterfall Models

- Iterative Waterfall Model
- Incremental Model
- Prototyping Model
- Spiral Method
- Agile development

WATERFALL MODEL

The Waterfall Model is a **LINEAR SEQUENTIAL MODEL**. In which progress is seen as flowing steadily downwards (like a waterfall) through the phases of software implementation. This means that any phase in the development process begins only if the previous phase is complete. The waterfall approach does not define the process to go back to the previous phase to handle changes in requirement. The waterfall approach was the earliest approach and most widely known that was used for software development.

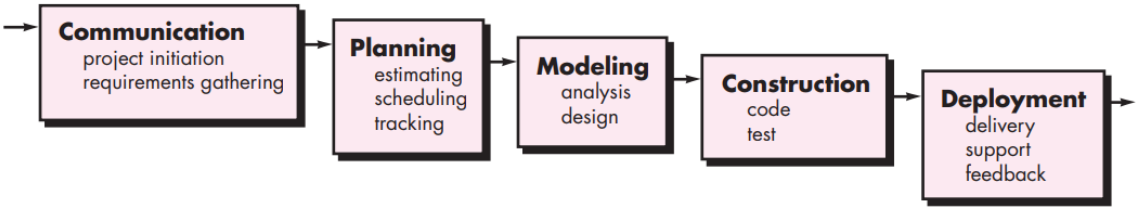

Figure: Generic Waterfall Model

Phases of Waterfall model: All work flows from **communication** towards **deployment** in a reasonably linear fashion.

- **Communication**: includes **project initiation** and **requirements gathering** activities.
- **Planning:** includes **estimating**, **scheduling** and **tracking** activities.
- **Modeling:** includes **analysis** and **design** activities.

- **Construction:** includes **coding** and **testing** activities.

- **Deployment:** includes product **delivery**, **support** and **feedback** activities.

ITERATIVE WATERFALL MODEL/ WATERFALL MODEL WITH FEEDBACK

To overcome the major shortcomings of the classical waterfall model, we come up with the iterative waterfall model.

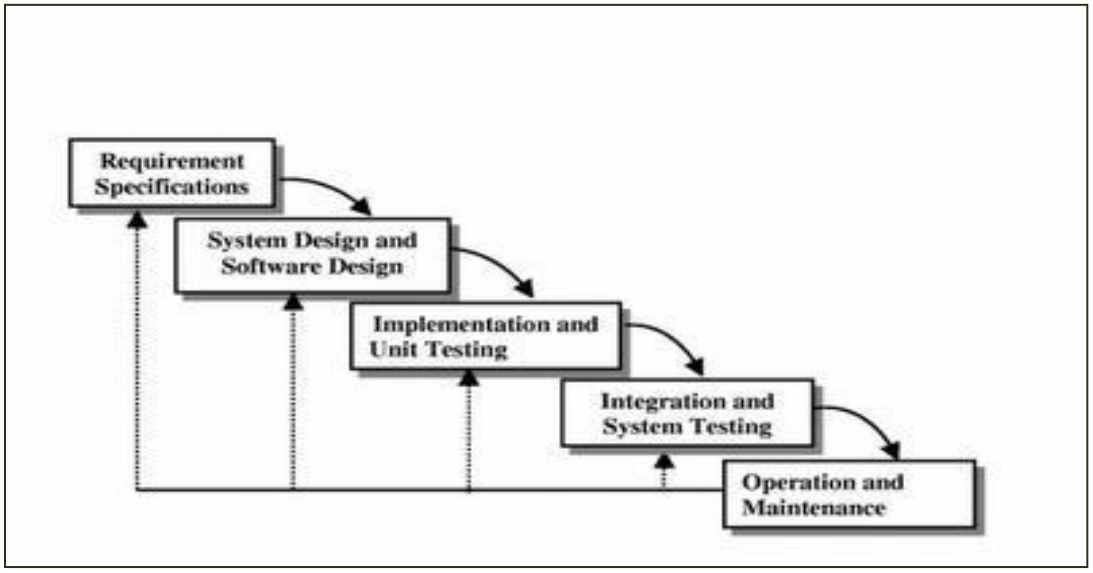

Figure : Iterative Waterfall Model

Here, we provide feedback paths for error correction as & when detected later in a phase. Though errors are inevitable, but it is desirable to detect them in the same phase in which they occur. If so, this can reduce the effort to correct the bug.

The advantage of this model is that there is a working model of the system at a very early stage of development which makes it easier to find functional or design flaws. Finding

issues at an early stage of development enables to take corrective measures in a limited budget.

The disadvantage with this SWDLC model is that it is applicable only to large and bulky software development projects. This is because it is hard to break a small software system into further small serviceable increments/modules.

THE V-MODEL

A variation in the representation of the waterfall model is called the V-model. Represented in Figure, the V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities.

As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution.

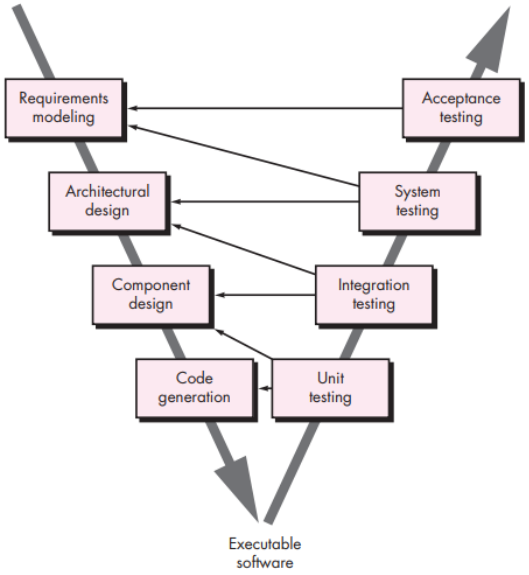

FIGURE: V- MODEL

Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side.

In reality, there is no fundamental difference between the classic waterfall life cycle and the V-model.

The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

The problems encountered when the waterfall model is applied are:

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

2. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

EVOLUTIONARY PROCESS MODELS

INCREMENTAL PROCESS MODELS

There are many situations in which initial software requirements are well defined, but it is not possible to follow a purely linear process. In addition, there may be a need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases. In such cases, you can choose a process model that is designed to produce the software in increments.

The incremental model combines elements of linear and parallel process flows and applies linear sequences in a stepwise manner according to calendar. **Each linear sequence produces deliverable "*increments*"** of the software.

For example, word-processing software developed using the incremental model may deliver:

- basic file management, editing, and document production functions in the **first increment**;
- more sophisticated editing and document production capabilities in the **second increment**;
- spelling and grammar checking in the **third increment**;
- and advanced page layout capability in the **fourth increment**.

It should be noted that the process flow for any increment can incorporate the ***prototyping methods.***

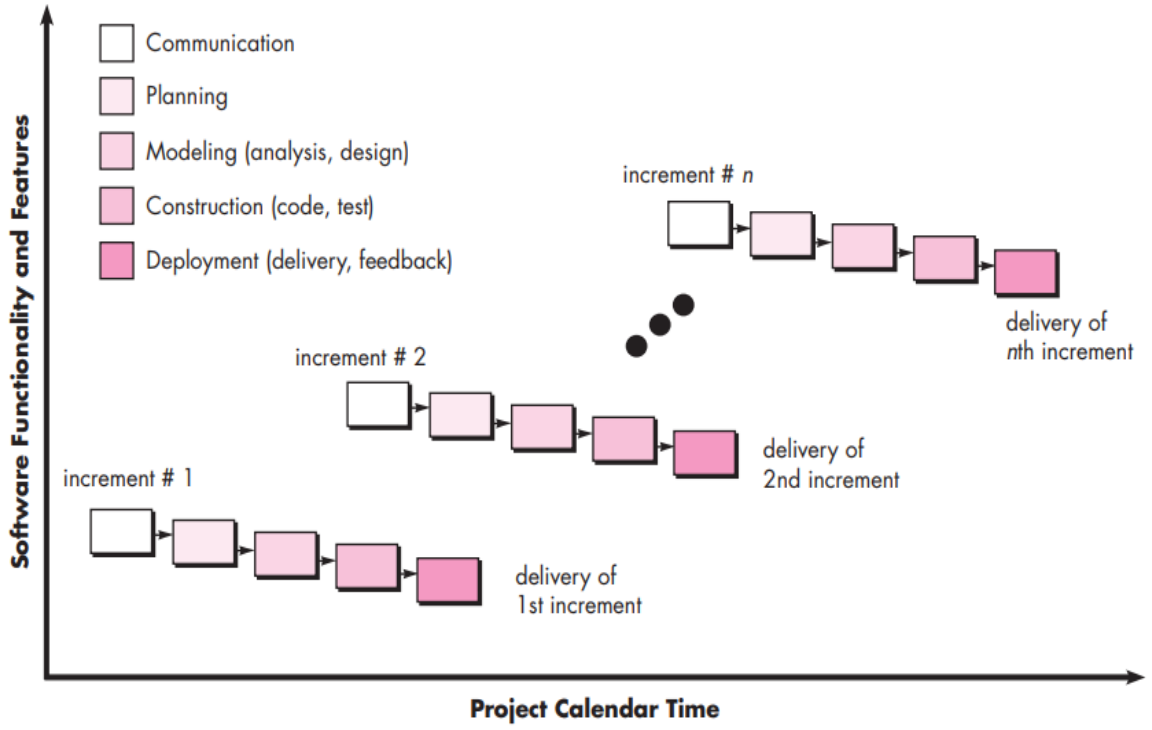

Figure: The Incremental Model

When an incremental model is used, the **first increment is often a core product**. That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation). As a result of use and/or evaluation, a plan is developed for the next increment.

The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project.

Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks.

For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end users without inordinate delay.

PROTOTYPING MODEL

Prototyping: Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form

that human-machine interaction should take. In these, and many other situations, a prototyping paradigm may offer the best approach.

Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models. Regardless of the manner in which it is applied, the prototyping paradigm assists you and other stakeholders to better understand what is to be built when requirements are fuzzy.

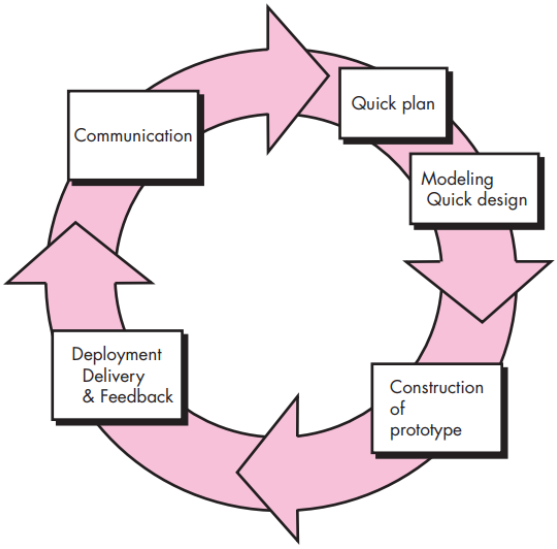

Figure: Prototyping

The prototyping paradigm begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.

A prototyping iteration is planned quickly, and modeling (in the form of a "quick design") occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats).

The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.

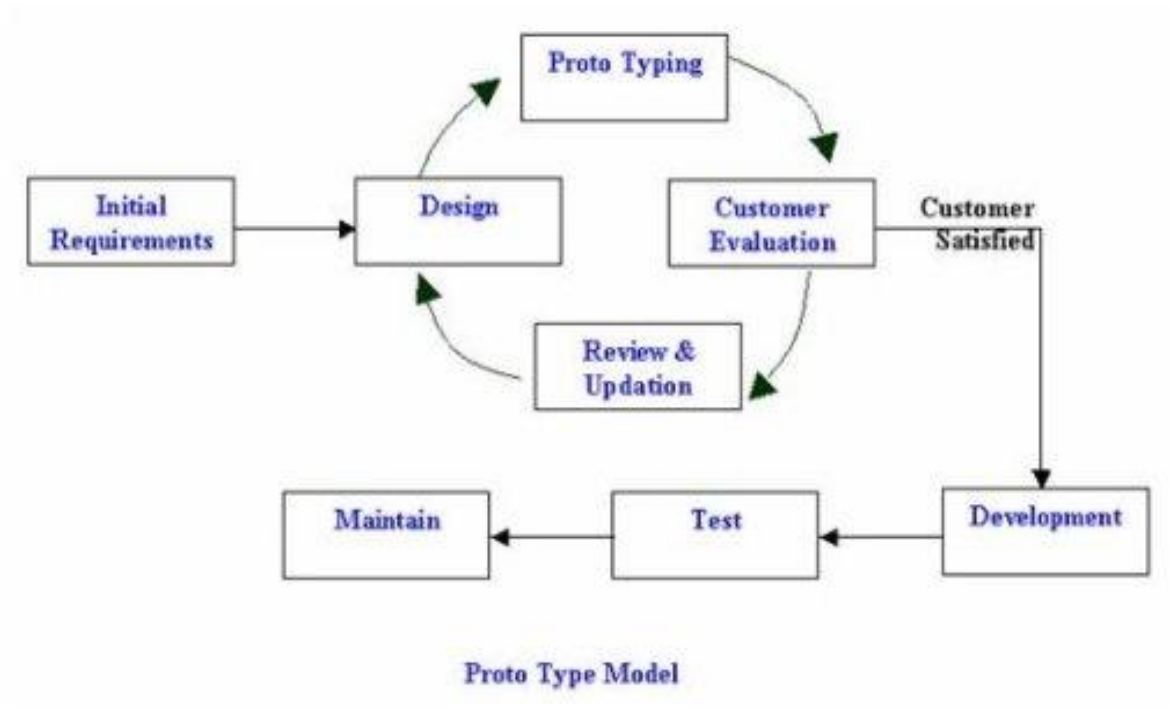

Figure: Prototype Model

Some problems may occur in Prototyping for the following reasons:

1. Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high levels of quality can be maintained, stakeholders cry foul and demand that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.

2. As a software engineer, you often make implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, you may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

Although these problems can occur, yet prototyping can be an effective method for software engineering.

THE SPIRAL MODEL

Originally proposed by Barry Boehm, the spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.

It provides the potential for rapid development of increasingly more complete versions of the software. Boehm describes the model in the following manner:

"The spiral development model is a risk-driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions."

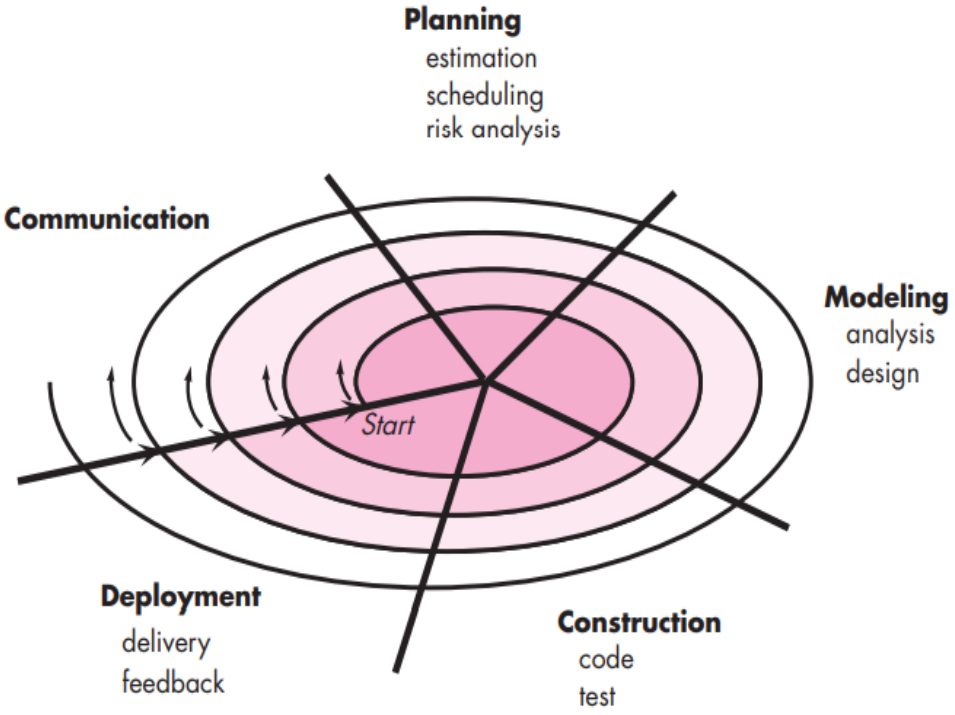

FIGURE: TYPICAL SPIRAL MODEL

[NOTE: The arrows pointing inward along the axis separating the deployment region from the communication region indicate a potential for local iteration along the same spiral path.]

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a set of **framework activities** defined by the software engineering team. Each of the framework activities represent one **segment** of the spiral path illustrated in Figure. The spiral model can be adapted to apply throughout the life of the computer software.

SEGMENTS:

SEGMENT 1 includes following activities: **Communication** (requirement gathering, customer evaluation and understanding)

SEGMENT 2 includes following activities: **Planning** (estimation, scheduling and risk analysis)

SEGMENT 3 includes following activities: **Modeling** (analysis and design)

SEGMENT 4 includes following activities: **Construction** (coding and testing)

SEGMENT 5 includes following activities: **Deployment** (delivery and feedback)

CIRCUITS AROUND THE SPIRAL:

As this evolutionary process begins, the software team performs activities that are implied by a **circuit** around the spiral in a clockwise direction, beginning at the center.

Risk is considered as each revolution is made.

Anchor point milestones, a combination of work products and conditions that are attained along the path of the spiral, are noted for each evolutionary **pass**.

The **first circuit** around the spiral might result in the development of a product specification and concept development of project, that starts at the core of the spiral and continues for multiple iterations until concept development is complete.

subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.

Each pass through the planning region results in adjustments to the project plan.

Cost and schedule are adjusted based on feedback derived from the customer after delivery.

In addition, the project manager adjusts the planned number of iterations required to complete the software.

The **version** or **build** or **deliverable** produced at the end of Deployment phase of **the last circuit**, is the **final software product.**

SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

A software requirements specification (SRS) is a detailed description of a software system to be developed with its functional and non-functional requirements.

The SRS is developed based the agreement between customer and contractors. It may include the use cases of how user is going to interact with software system. The software requirement specification document consistent of all necessary requirements required for project development.

To develop the software system we should have clear understanding of Software system. To achieve this we need to continuous communication with customers to gather all requirements.

A good SRS defines the how Software System will interact with all internal modules, hardware, communication with other programs and human user interactions with wide range of real life scenarios.

Using the Software requirements specification (SRS) document on QA lead, managers creates test plan. It is very important that testers must be cleared with every detail specified in this document in order to avoid faults in test cases and its expected results.

It is highly recommended to review or test SRS documents before start writing test cases and making any plan for testing.

Let's see how to test SRS and the important point to keep in mind while testing it.

1. Correctness of SRS should be checked. Since the whole testing phase is dependent on SRS, it is very important to check its correctness. There are some standards with which we can compare and verify.

2. Ambiguity should be avoided. Sometimes in SRS, some words have more than one meaning and this might confused testers making it difficult to get the exact reference. It is advisable to check for such ambiguous words and make the meaning clear for better understanding.

3. Requirements should be complete. When tester writes test cases, what exactly is required from the application, is the first thing which needs to be clear. For e.g. if application needs to

send the specific data of some specific size then it should be clearly mentioned in SRS that how much data and what is the size limit to send.

4. Consistent requirements. The SRS should be consistent within itself and consistent to its reference documents. If you call an input "Start and Stop" in one place, don't call it "Start/Stop" in another. This sets the standard and should be followed throughout the testing phase.

5. Verification of expected result: SRS should not have statements like "Work as expected", it should be clearly stated that what is expected since different testers would have different thinking aspects and may draw different results from this statement.

6. Testing environment: some applications need specific conditions to test and also a particular environment for accurate result. SRS should have clear documentation on what type of environment is needed to set up.

7. Pre-conditions defined clearly: one of the most important part of test cases is pre-conditions. If they are not met properly then actual result will always be different expected result. Verify that in SRS, all the pre-conditions are mentioned clearly.

8. Requirements ID: these are the base of test case template. Based on requirement Ids, test case ids are written. Also, requirements ids make it easy to categorize modules so just by looking at them, tester will know which module to refer. SRS must have them such as id defines a particular module.

9. Security and Performance criteria: security is priority when a software is tested especially when it is built in such a way that it contains some crucial information when leaked can cause harm to business. Tester should check that all the security related requirements are properly defined and are clear to him. Also, when we talk about performance of a software, it plays a very important role in business so all the requirements related to performance must be clear to the tester and he must also know when and how much stress or load testing should be done to test the performance.

10. Assumption should be avoided: sometimes when requirement is not cleared to tester, he tends to make some assumptions related to it, which is not a right way to do testing as assumptions could go wrong and hence, test results may vary. It is better to avoid

assumptions and ask clients about all the "missing requirements" to have a better understanding of expected results.

11. Deletion of irrelevant requirements: there are more than one team who work on SRS so it might be possible that some irrelevant requirements are included in SRS. Based on the understanding of the software, tester can find out which are these requirements and remove them to avoid confusions and reduce work load.

12. Freeze requirements: when an ambiguous or incomplete requirement is sent to client to analyze and tester gets a reply, that requirement result will be updated in the next SRS version and client will freeze that requirement. Freezing here means that result will not change again until and unless some major addition or modification is introduced in the software.

Most of the defects which we find during testing are because of either incomplete requirements or ambiguity in SRS. To avoid such defects it is very important to test software requirements specification before writing the test cases. Keep the latest version of SRS with you for reference and keep yourself updated with the latest change made to the SRS. Best practice is to go through the document very carefully and note down all the confusions, assumptions and incomplete requirements and then have a meeting with the client to get them clear before development phase starts as it becomes costly to fix the bugs after the software is developed. After all the requirements are cleared to a tester, it becomes easy for him to write effective test cases and accurate expected results.

FORMAL Requirement SPECIFICATION

A formal software specification is a statement expressed in a language whose vocabulary, syntax, and semantics are formally defined. The need for a formal semantic definition means that the specification languages cannot be based on natural language; it must be based on mathematics.

The advantages of a formal language are:

• The development of a formal specification provides insights and understanding of the software requirements and the software design.

• Given a formal system specification and a complete formal programming language definition, it may be possible to prove that a program conforms to its specifications.

• Formal specification may be automatically processed. Software tools can be built to assist with their development, understanding, and debugging.

• Depending on the formal specification language being used, it may be possible to animate a formal system specification to provide a prototype system.

• Formal specifications are mathematical entities and may be studied and analyzed using mathematical methods.

• Formal specifications may be used as a guide to the tester of a component in identifying appropriate test cases.

Relational and State-Oriented Notations

Relational notations are used based on the concept of entities and attributes.

Entities are elements in a system; the names are chosen to denote the nature of the elements (e.g., stacks, queues).

Attributes are specified by applying functions and relations to the named entities.

Attributes specify permitted operations on entities, relationships among entities, and data flow between entities.

Relational notations include implicit equations, recurrence relations, and algebraic axioms. State-oriented specifications use the current state of the system and the current stimuli presented to the system to show the next state of the system.

The execution history by which the current state was attained does not influence the next state; it is dependent only on the current state and the current stimuli.

State-oriented notations include decision tables, event tables, transition tables, and finite-state tables.

SPECIFICATION PRINCIPLES

Principle 1: Separate functionality from implementation. A specification is a statement of what is desired, not how it is to be realized. Specifications can take two general forms. The first form is that of mathematical functions: Given some set of inputs, produce a particular set of outputs. The general form of such specifications is find [a/the/all] result such that P(input), where P represents an arbitrary predicate. In such specifications, the result to be obtained has been entirely expressed in a "what", rather than a "how" form, mainly because the result is a mathematical function of the input (the operation has well-defined starting and stopping points) and is unaffected by any surrounding environment.

Principle 2: A process-oriented systems specification language is sometimes required. If the environment is dynamic and its changes affect the behavior of some entity interacting with that environment (as in an embedded computer system), its behavior cannot be expressed as a mathematical function of its input. Rather a process-oriented description must be employed, in which the "what" specification is achieved by specifying a model of the desired behavior in terms of functional responses to various stimuli from the environment.

Principle 3: The specification must provide the implementer all of the information he/she needs to complete the program, and no more. In particular, no information about the structure of the calling program should be conveyed.

Principle 4: The specification should be sufficiently formal that it can conceivably be tested for consistency, correctness, and other desirable properties.

Principle 5: The specification should discuss the program in terms normally used by the user and implementer alike.

SOME SPECIFICATION TECHNIQUES

1. Implicit Equations

Specify computation of square root of a number between 0 and some maximum value Y to a tolerance E.

(0<=X<=Y){ABS_VALUE[(WHAT(X)) 2-X]}<=E

2. Recurrence Relation

Good for recursive computations.

Example, Fibonacci numbers 0, 1, 1, 2, 3, 5, 8,...

FI(0) = 0;

FI(1) = 1;

FI(n) = FI(n-1) + FI(n-2); for n>=1.

Verification And Validation

We have seen the "V-Model". In the V Model Software Development Life Cycle, based on requirement specification document the development & testing activity is started.

The V-model is also called as Verification and Validation model.

The testing activity is perform in the each phase of Software Testing Life Cycle.

In the first half of the model validations testing activity is integrated in each phase like review user requirements, System Design document & in the next half the Verification testing activity is come in picture.

Verification ***(ARE WE BUILDING THE PRODUCT RIGHT?)***

Definition *: The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.*

Verification is a static practice of verifying documents, design, code and program. It includes all the activities associated with producing high quality software: inspection, design analysis and specification analysis. It is a relatively objective process.

Verification will help to determine whether the software is of high quality, but it will not ensure that the system is useful. Verification is concerned with whether the system is well-engineered and error-free.

Methods of Verification : Static Testing

- *Walkthrough*
- *Inspection*
- *Review*

Validation ***(ARE WE BUILDING THE RIGHT PRODUCT?)***

Definition*: The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.*

Validation is the process of evaluating the final product to check whether the software meets the customer expectations and requirements. It is a dynamic mechanism of validating and testing the actual product.

Methods of Validation : Dynamic Testing

- *Testing according to End Users*

Difference between Verification and Validation

The distinction between the two terms is largely to do with the role of specifications.

Verification *is the process of checking that the software meets the specification. "Did I build what I need?"*

Validation *is the process of checking whether the specification captures the customer's needs. "Did I build what I said I would?"*

Verification	***Validation***
Verification is a static practice of verifying documents, design, code and program.	Validation is a dynamic mechanism of validating and testing the actual product.
It does not involve executing the code.	It always involves executing the code.
It is human based checking of documents and files.	It is computer based execution of program.
Verification uses methods like inspections, reviews, walkthroughs, and Desk-checking etc.	Validation uses methods like black box (functional) testing, gray box testing, and white box (structural) testing etc.
Verification is to check whether the software conforms to specifications.	Validation is to check whether software meets the customer expectations and requirements.
It can catch errors that validation cannot catch. It is low level exercise.	*It can catch errors that verification cannot catch. It is High Level Exercise.*
Target is requirements specification, application and software architecture, high level, complete design, and database design etc.	Target is actual product-a unit, a module, a bent of integrated modules, and effective final product.

Verification is done by QA team to ensure that the software is as per the specifications in the SRS document.	***Validation*** *is carried out with the involvement of testing team.*
It generally comes first-done before **validation**.	It generally follows after **verification**.
Are we building the system right?	Are we building the right system?
Verification is the process of evaluating products of a development phase to find out whether they meet the specified requirements.	Validation is the process of evaluating software at the end of the development process to determine whether software meets the customer expectations and requirements.
The objective of Verification is to make sure that the product being develop is as per the requirements and design specifications.	The objective of Validation is to make sure that the product actually meet up the user's requirements, and check whether the specifications were correct in the first place.
Following activities are involved in Verification: Reviews, Meetings and Inspections.	Following activities are involved in Validation: Testing like black box testing, white box testing, gray box testing etc.
Verification is carried out by QA team to check whether implementation software is as per specification document or not.	Validation is carried out by testing team.
Execution of code is not comes under Verification.	Execution of code is comes under Validation.
Verification process explains whether the outputs are according to inputs or not.	Validation process describes whether the software is accepted by the user or not.

Verification is carried out before the Validation.	Validation activity is carried out just after the Verification.
Following items are evaluated during Verification: Plans, Requirement Specifications, Design Specifications, Code, Test Cases etc,	Following item is evaluated during Validation: Actual product or Software under test.
Cost of errors caught in Verification is less than errors found in Validation.	Cost of errors caught in Validation is more than errors found in Verification.
It is basically manually checking the of documents and files like requirement specifications etc.	It is basically checking of developed program based on the requirement specifications documents & files.