1

INTRODUCTION

Unit Structure :

1.0 OBJECTIVES

After going through this chapter, you will be able to:

- Software
- Software Engineering
- Different Process Models used in software Engineering

1.1 Introduction

The end product that software developers create and provide ongoing support for is computer software. It includes computer programmes that run on machines of every size and architecture, content displayed while computer programmes run, and descriptive data in both physical and digital formats that cover almost any electronic medium.Software engineers can create high-quality computer software through the use of a methodology, a set of techniques, and a variety of tools.

1.2 THE NATURE OF SOFTWARE

Software serves two functions. It is both a product and a vehicle used to transport a product. As a product, it provides the processing power embodied by computer hardware or, more broadly, by a network of computers accessible via local hardware. Software is an information transformer, whether it is found in a mobile phone or a mainframe computer. It produces, manages, acquires, modifies, displays, or transmits data.As the vehicle used to deliver the product, software serves as the foundation for computer control (operating systems), information communication (networks), as well as the creation and control of additional programs (software tools and environments).

1.2.1 Defining Software

- Software is instructions (computer programs) that when executed provide desired features, function, and performance.
- Software is developed or engineered; it is not manufactured in the classical sense
- **Software doesn't "wear out".**
    - When something is no longer of any use, it reaches the "wear out" state. That is, it can not perform the function it was built for. For example, a printer reaches "wear out" state and it can't print anymore. This doesn't include the recycling options. One makes use of a dead printer to do anything else but printing.
    - On the other hand, software does not wear out. Like hardware, software also shows a high failure rate at its infant state. Then it gets modifications and the defects get corrections and thus it comes to the idealized state. This idealized state continues.
    - alternative software with implementation of current user demands can replace a software. Though, not having a recent feature is not a defect, users tend to use the latest alternatives. If we consider this as failure for the software then the failure rate increases with time. This will make the software deteriorate due to change, but still the software can perform it's operation as it was performing in the beginning. That is why software doesn't wear out.

1.2.2 Software Application Domain

There are seven categories of software

1. **System software**:is a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate,information structures. Other systems applications **(e.g Operating system components, drivers, networking software, telecommunications processors)** process largely indeterminate data.
2. Application software:is a stand-alone program that solves a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making.One of the significant and

essential things to note about application software is that it cannot run independently. To run application software, you have to use a system platform capable of supporting it.

3. **Engineering/scientific software**:has been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing.

4. **Embedded software**:resides within a product or system and is used to implement and control features and functions for the end user and for the system itself.**e.g., keypad control for a microwave oven, digital functions in an automobile such as fuel control, dashboard displays, and braking systems.**

5. **Product-line software:**It is designed to provide a specific capability for use by many different customers. Product-line software can focus mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications).

6. **Web applications**:It is called "WebApps," this network-centric software category spans a wide array of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics.

7. **Artificial intelligence software**—makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

1.3 SOFTWARE ENGINEERING

- Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.

- Software Engineering is defined as the systematic approach to the development, operation, maintenance, and retirement of software.

- Software engineering is a layered technology. Referring to Figure 1.1, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management, Six Sigma, and similar philosophies10 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.

- **Quality focus:** The bedrock that supports software engineering is a **quality focus**.

- **Process Layer**:The foundation for software engineering is the **process layer**. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework that must be established for effective delivery of software engineering technology. The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

- **Method:**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.

- **Tools:**Software engineering tools provide automated or semi automated 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

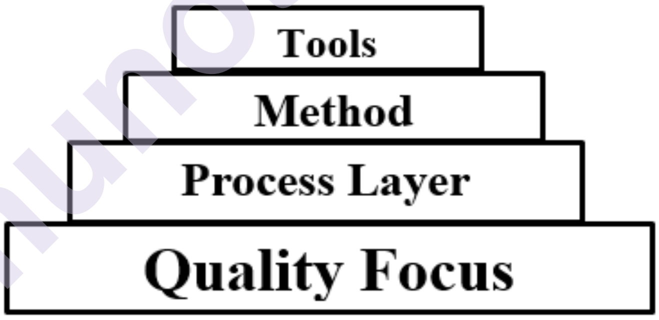


Fig 1.1 Software Engineering Layers

1.4 SOFTWARE PROCESS

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

- 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 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.

Process Framework Activities:

For the purpose of illustrating typical process activities, the process framework is necessary. A process framework for software engineering lists five framework tasks. Framework activities include, for instance, planning, modeling, building, and implementation. A set of required work outputs, project milestones, and software quality assurance (SQA) points are included in each engineering action specified by a framework activity.

- **Communication:** By communication, customer requirement gathering is done. Communication with consumers and stakeholders to determine the system's objectives and the software's requirements.

- **Planning:** Establish engineering work plan, describes technical risk, lists resources requirements, work produced and defines work schedule.

- **Modeling:** Architectural models and design to better understand the problem and for work towards the best solution. The software model is prepared by:
  o Analysis of requirements
  o Design

- **Construction:** Creating code, testing the system, fixing bugs, and confirming that all criteria are met. The software design is mapped into a code by:
  o Code generation
  o Testing

- **Deployment:** In this activity, a complete or non-complete product or software is represented to the customers to evaluate and give feedback. On the basis of their feedback, we modify the product for the supply of better products.

Umbrella Activities:

Software engineering process framework activities are complemented by a number of umbrella activities. In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk. Typical umbrella activities include:

- **Software project tracking and control**—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule. Risk management—assesses risks that may affect the outcome of the project or the quality of the product.

- **Software quality assurance**—defines and conducts the activities required to ensure software quality.

- **Technical reviews**—assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.
- **Measurement**—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.
- **Software configuration management**—manages the effects of change throughout the software process.
- **Reusability management**—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.
- **Work product preparation and production**—encompasses the activities required to create work products such as models, documents, logs, forms, and lists

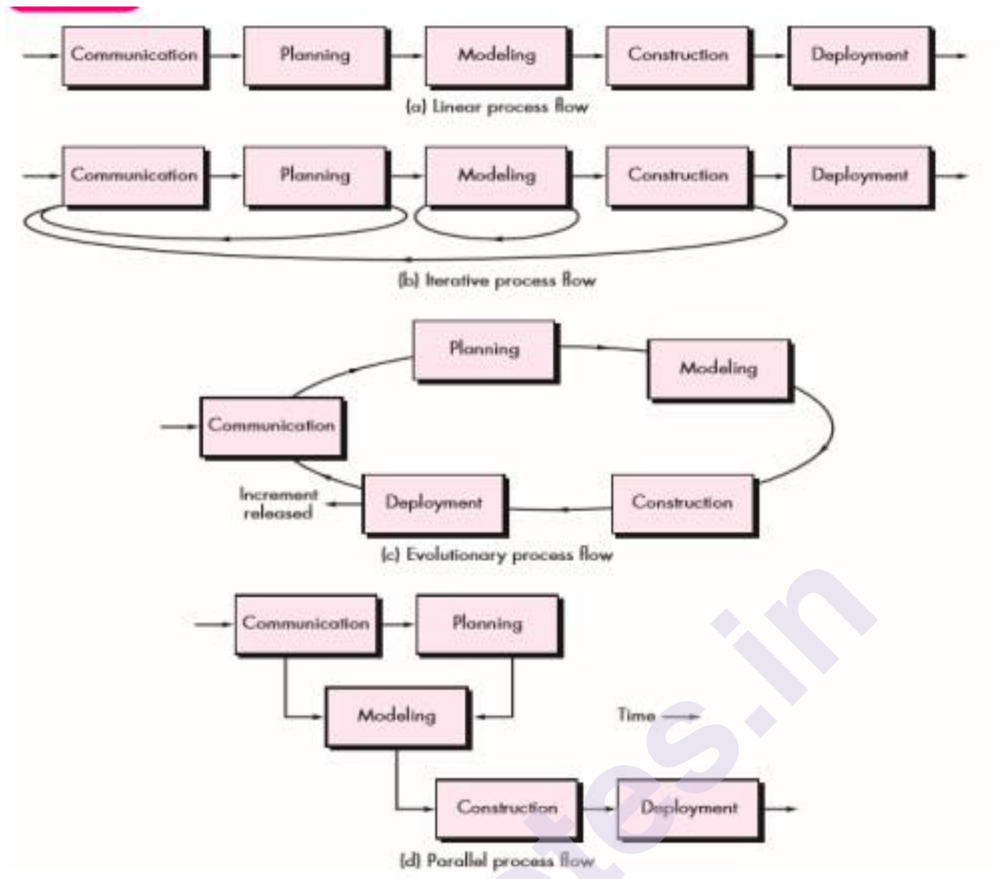
1.5 GENERIC PROCESS

A process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another. Each software engineering action is defined by a task set that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

A generic process framework for software engineering defines five framework activities— communication, planning, modeling, construction, and deployment. In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process.

1. A linear process flow executes each of the five framework activities in sequence, beginning with communication and culminating with deployment (Figure 1.2 a).
2. An iterative process flow repeats one or more of the activities before proceeding to the next (Figure 1.2b).
3. An evolutionary process flow executes the activities in a "circular" manner. (Figure 1.2c).
4. A parallel process flow (Figure 1.2d) executes one or more activities in parallel with other activities.

Fig 1.2 Process Flow



Identifying a Task Set

- First, choose a task set that best accommodates the needs of the project and the characteristics of your team.
- A task set defines the actual work to be done to accomplish the objectives of a software engineering action.
    - ❖ A list of the task to be accomplished
    - ❖ A list of the work products to be produced
    - ❖ A list of the quality assurance filters to be applied

Process Pattern

- A process pattern describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem.
- In more general terms, a process pattern provides us with a template, a consistent method for describing problem solutions within the context of the software process. By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.

- Process pattern types-

  Stage patterns — defines a problem associated with a framework activity for the process.

  Task patterns — defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice

  Phase patterns — define the sequence of framework activities that occur with the process, even when the overall flow of activities is iterative in nature.

1.6 THE WATERFALL MODEL

The waterfall model, sometimes called the classic life cycle, suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment.

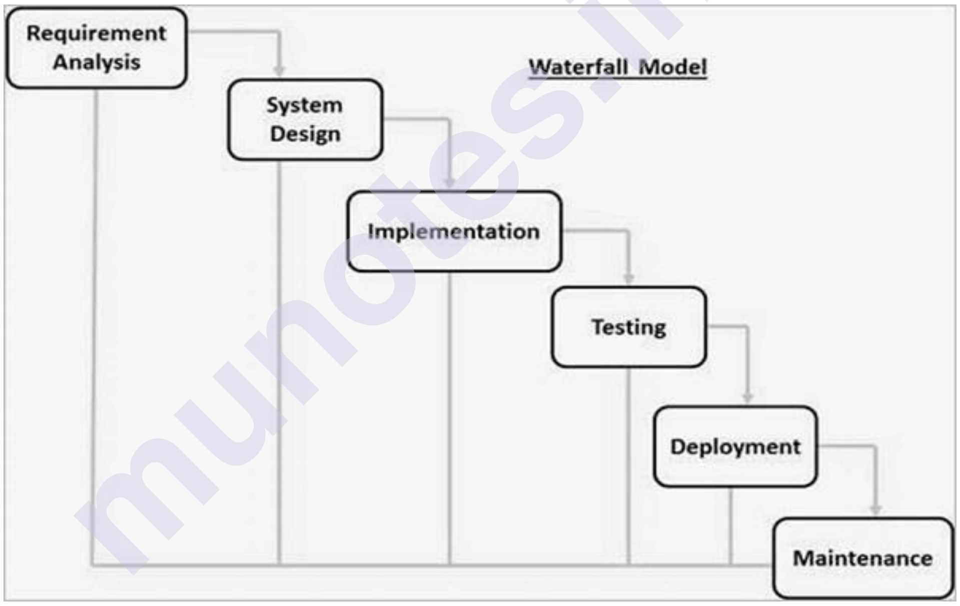


Fig 1.3 WaterFall Model

A variation in the representation of the waterfall model is called the V-model. Represented in Figure 1.4. 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. 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.

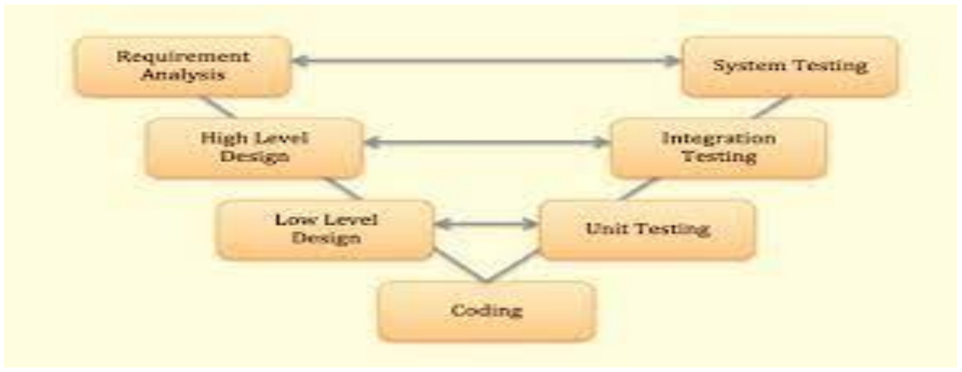


Fig 1.4 V-model

Advantages of waterfall model-

- This model works for small projects because the requirements are understood very well.
- The waterfall model is simple and easy to understand, implement, and use.
- All the requirements are known at the beginning of the project, hence it is easy to manage.

Disadvantages of the waterfall model

- The problems with this model are uncovered, until the software testing.
- The amount of risk is high.
- This model is not good for complex and object oriented projects.

1.7 INCREMENTAL PROCESS MODEL

The incremental model combines elements of linear and parallel process flows Referring to Figure 1.5, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software. 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.

Advantages of incremental model

- This model is flexible because the cost of development is low and initial product delivery is faster.
- It is easier to test and debug during the smaller iteration.
- The working software generates quickly and early during the software life cycle. •
- The customers can respond to its functionalities after every increment.

Disadvantages of the incremental model

- The cost of the final product may cross the cost estimated initially.
- This model requires very clear and complete planning.
- The planning of design is required before the whole system is broken into small increments.
- The demands of customer for the additional functionalities after every increment causes problem during the system architecture.

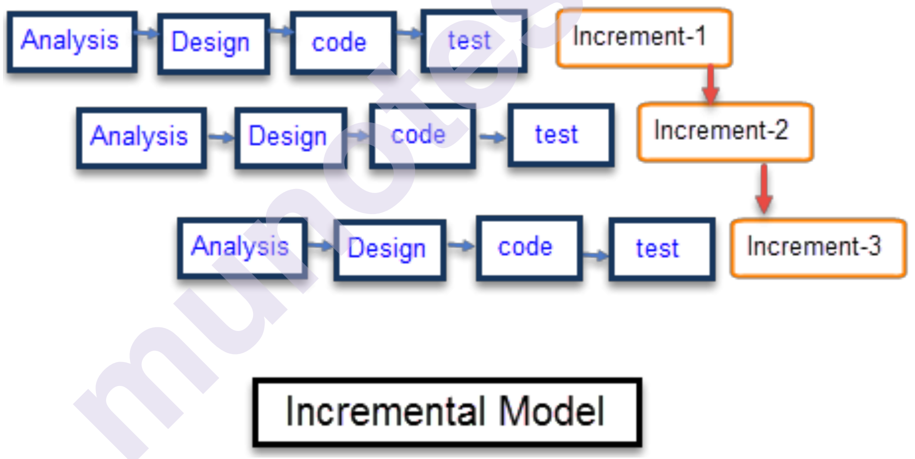


Fig 1.5 Incremental Model

1.8 EVOLUTIONARY PROCESS MODELS

Evolutionary models are iterative type models.They allow to develop more complete versions of the software. Following are the evolutionary process models.

1. The prototyping model
2. The spiral model
3. Concurrent development model

1.8.1. The Prototyping model

- Prototype is defined as the first or preliminary form using which other forms are copied or derived.
- Prototype model is a set of general objectives for software.It does not identify the requirements like detailed input, output.
- It is a software working model of limited functionality.In this model, working programs are quickly produced.

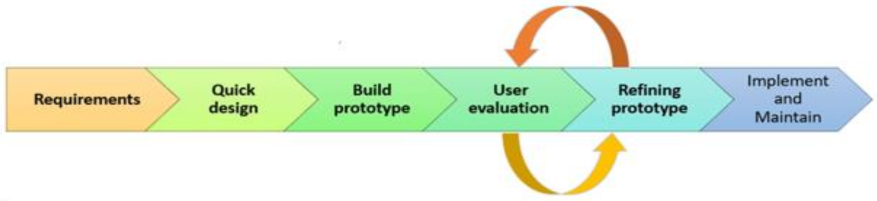


Fig1.6 Prototyping Model

The different phases of Prototyping model are

- **Communication:**

  In this phase, developers and customers meet and discuss the overall objectives of the software.

- **Quick design**

  Quick design is implemented when requirements are known.It includes only the important aspects like input and output format of the software.It focuses on those aspects which are visible to the user rather than the detailed plan.It helps to construct a prototype.

- **Modeling quick design**

  This phase gives a clear idea about the development of software because the software is now built.It allows the developer to better understand the exact requirements.

- **Construction of prototype**

  The prototype is evaluated by the customer itself.

- **Deployment, delivery, feedback**

  If the user is not satisfied with the current prototype then it refines according to the requirements of the user.The process of refining the prototype is repeated until all the requirements of users are met. When the users are satisfied with the developed prototype then the system is developed on the basis of final prototype.

Advantages of Prototyping Model:

- Prototype models need not know the detailed input, output, processes, adaptability of the operating system and full machine interaction.
- In the development process of this model users are actively involved.
- The development process is the best platform to understand the system by the user.
- Errors are detected much earlier.
- Gives quick user feedback for better solutions.
- It identifies the missing functionality easily.
- It also identifies the confusing or difficult functions.

Disadvantages of Prototyping Model:

- The client involvement is more and it is not always considered by the developer.
- It is a slow process because it takes more time for development.
- Many changes can disturb the rhythm of the development team.
- It is a thrown away prototype when the users are confused with it.

1.8.2. The Spiral model

Spiral model is a risk driven process model.It is used for generating software projects.

In a spiral model, an alternate solution is provided if the risk is found in the risk analysis, then alternate solutions are suggested and implemented.

It is a combination of prototype and sequential model or waterfall model.

In one iteration all activities are done, for large project's the output is small.

The framework activities of the spiral model are as shown in the following figure.

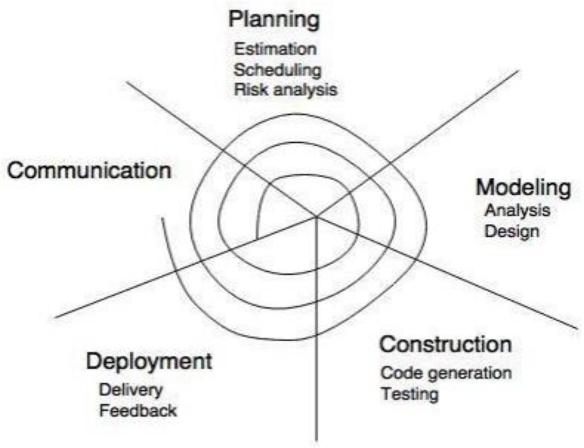


Fig. - The Spiral Model

NOTE: The description of the phases of the spiral model is same as that of the process model.

Advantages of Spiral Model

- It reduces a high amount of risk.
- It is good for large and critical projects.
- It gives strong approval and documentation control.
- In the spiral model, the software is produced early in the life cycle process.

Disadvantages of Spiral Model

- It can be costly to develop a software model.
- It is not used for small projects.

1.8.3. The concurrent development model

- The concurrent development model is called a concurrent model.
- The communication activity has completed in the first iteration and exits in the awaiting changes state.
- The modeling activity completed its initial communication and then went to the underdevelopment state.
- If the customer specifies the change in the requirement, then the modeling activity moves from the under development state into the awaiting change state.
- The concurrent process models activities moving from one state to another state.

Advantages of the concurrent development model

- This model is applicable to all types of software development processes.
- It is easy to understand and use.
- It gives immediate feedback from testing.
- It provides an accurate picture of the current state of a project.

Disadvantages of the concurrent development model

- It needs better communication between the team members.
- This may not be achieved all the time.
- It requires us to remember the status of the different activities.

1.8.4 Component Based Models

- Component based development is a software system development methodology where the system is developed using reusable software components. Component based development aims at improved efficiency, performance and quality of the system by recycling components.
- Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built.
- The component-based development model incorporates many of the characteristics of the spiral model.The component-based development model incorporates the following steps (implemented using an evolutionary approach):

1. Available component-based products are researched and evaluated for the application domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.
4. Components are integrated into the architecture.
5. Comprehensive testing is conducted to ensure proper functionality

1.9 UNIFIED PROCESS MODEL

The life of a software system can be represented as a series of **cycles**. A cycle ends with the release of a version of the system to customers.Within the Unified Process, each cycle contains Five phases. A **phase** is simply the span of time between two **major milestones**, points at which managers make important decisions about whether to proceed with development and, if so, what's required concerning project scope, budget, and schedule.

Inception

The primary goal of the **Inception phase** is to establish the case for the viability of the proposed system.

The tasks that a project team performs during Inception include the following:

- Defining the scope of the system (that is, what's in and what's out)
- Outlining a **candidate architecture**, which is made up of initial versions of six different models
- Identifying critical risks and determining when and how the project will address them
- Starting to make the business case that the project is worth doing, based on initial estimates of cost, effort, schedule, and product quality

Elaboration

The primary goal of the **Elaboration phase** is to establish the ability to build the new system given the financial constraints, schedule constraints, and other kinds of constraints that the development project faces.

The tasks that a project team performs during Elaboration include the following:

- Capturing a healthy majority of the remaining functional requirements
- Expanding the candidate architecture into a full **architectural baseline**, which is an internal release of the system focused on describing the architecture
- Addressing significant risks on an ongoing basis
- Finalizing the business case for the project and preparing a project plan that contains sufficient detail to guide the next phase of the project (Construction)

Construction

- The primary goal of the **Construction phase** is to build a system capable of operating successfully in beta customer environments.
- During Construction, the project team performs tasks that involve building the system iteratively and incrementally (see "Iterations and Increments" later in this chapter), making sure that the viability of the system is always evident in executable form.
- The major milestone associated with the Construction phase is called **Initial Operational Capability**. The project has reached this milestone if a set of beta customers has a more or less fully operational system in their hands.

Transition

- The primary goal of the **Transition phase** is to roll out the fully functional system to customers.
- During Transition, the project team focuses on correcting defects and modifying the system to correct previously unidentified problems.
- The major milestone associated with the Transition phase is called **Product Release**.

Production

The production phase of the UP coincides with the deployment activity of the generic process. During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.

1.10 AGILE DEVELOPMENT- AGILITY

- Agility means effective (rapid and adaptive) response to change, effective communication among all stockholders.
- Drawing the customer onto a team and organizing a team so that it is in control of work performed.
- The agile process forces the development team to focus on software itself rather than design and documentation.
- The agile process believes in iterative methods.
- The aim of agile process is to deliver the working model of software quickly to the customer For example: Extreme programming is the best known of agile process.
- Agility can be applied to any software process. However, to accomplish this, it is essential that the process be designed in a way that allows the project team to adapt tasks and to streamline them, conduct planning in a way that understands the fluidity of an agile development approach, eliminate all but the most essential work products and keep them lean, and emphasize an incremental delivery strategy that gets working software to the customer as rapidly as feasible for the product type and operational environment.

1.11 AGILE PROCESS

Any agile software process is characterized in a manner that addresses a number of key assumptions about the majority of software projects:

1. It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.
2. For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

Agility Principles

Agility principles for those who want to achieve agility:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4. Business people and developers must work together daily throughout the project.

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

7. Working software is the primary measure of progress.

8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Human Factors

If members of the software team are to drive the characteristics of the process that is applied to build software, a number of key traits must exist among the people on an agile team and the team itself

- **Competence:“**competence” encompasses innate talent, specific software-related skills, and overall knowledge of the process that the team has chosen to apply.

- **Common focus**:All team members should be focused on one goal—to deliver a working software increment to the customer within the time promised.

- **Collaboration**: Software engineering (regardless of process) is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team; and building information (computer software and relevant databases) that provides business value for the customer. To accomplish these tasks, team members must collaborate—with one another and all other stakeholders.

- **Decision-making ability**.:The team is given autonomy—decision-making authority for both technical and project issues.

- **Fuzzy problem-solving ability**. Software managers must recognize that the agile team will continually have to deal with ambiguity and will continually be buffeted by change.

- **Self-organization:**In the context of agile development, self-organization implies three things: (1) the agile team organizes itself for the work to be done, (2) the team organizes the process to best accommodate its local environment, (3) the team organizes the work schedule to best achieve delivery of the software increment.

1.12 EXTREME PROGRAMMING

- Extreme programming uses an object-oriented approach as its preferred development paradigm.
- Extreme programming encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing.

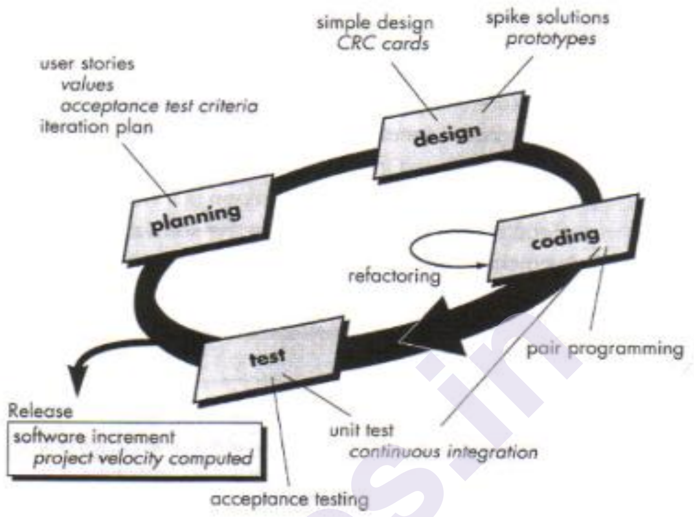


Fig. Extreme Programming Process

1. Planning:

- The planning activity begins with the creation of a set of stories that describe required features and functionality for software to be built.
- Each story is written by the customer and is placed on an index card. The customer assigns a value to the story based on the overall business value of the feature of function.
- Members of the XP (Extreme Programming) team then assess each story and assign a cost – measured in development weeks – to it.
- If the story will require more than three development weeks, the customer is asked to split the story into smaller stories, and the assignment of value and cost occurs again.
- Customers and the XP team work together to decide how to group stories into the next release to be developed by the XP team.
- Once a basic commitment is made for a release, the XP team orders the stories that will be developed in one of three ways:
    1. All stories will be implemented immediately.
    2. The stories with highest value will be moved up in the schedule and implemented first.
    3. The riskiest stories will be moved up in the schedule and implemented first.

- As development work proceeds, the customer can add stories, change the value of an existing story, split stories or eliminate them.
- The XP team then reconsiders all remaining releases and modifies its plan accordingly.

2. Design :

- XP design follows the KIS (Keep It Simple) principle. A simple design is always preferred over a more complex representation.
- The design provides implementation guidance for a story as it is written – nothing less, nothing more.
- XP encourages the use of CRC (Class Responsibility Collaborator) cards as an effective mechanism for thinking about the software in an object oriented context.
- CRC cards identify and organize the object oriented classes that are relevant to current software increment.
- The CRC cards are the only design work product produced as a part of XP process.
- If a difficult design is encountered as a part of the design of a story, XP recommends the immediate creation of that portion of the design called a 'spike solution'.
- XP encourages refactoring – a construction technique.

3. Coding

- XP recommends that after stories are developed and preliminary design work is done, the team should not move to cord, but rather develop a series of unit test that will exercise each story.
- Once the unit test has been created, the developer is better able to focus on what must be implemented to pass the unit test.
- Once the code completes, it can be unit tested immediately, thereby providing instantaneous feedback to the developer.
- A key concept during the coding activity is pair programming. XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real time problem solving and real time quality assurance.
- As pair programmers complete their work, the code they developed is integrated with the work of others.
- This continuous integration strategy helps to avoid compatibility and interfacing problems and provides a smoke testing environment that helps to uncover errors early.

4. Testing :

- The creation of unit tests before coding is the key element of the XP approach.
- The unit tests that are created should be implemented using a framework that enables them to be automated. This encourages regression testing strategy whenever code is modified.
- Individual unit tests are organized into a "Universal Testing Suit",

integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things are going away.

- XP acceptance tests, also called customer tests, are specified by the customer and focus on the overall system feature and functionality that are visible and reviewable by the customer.

LET US SUM UP

This chapter provides a clear Software . It covers different models used in software engineering. It also presents different unified process phases.The Second half of the chapter focuses on Agile Development.The chapter concludes with a clear overview of Extreme programming.

QUESTIONS

1.What are the nature of software.Explain

2.Explain the term “software doesn't wear out”

3.Explain Software Engineering

4.Explain Software Process

5.What are the umbrella activities involved in software process

6.Explain generic process

7.Explain waterfall Model

8.Explain Incremental Model,Iterative Model,prototyping Model

9.Explain Unified Process Model

10.what are Agility Principles?

11.Explain Extreme Programming.

REFERENCES:

- *https://technostacks.com/blog/types-of-application-software/*
- *http://stg-tud.github.io/sedc/Lecture/ws16-17/6-SPL.pdf*
- *https://edscl.in/pluginfile.php/1659/mod_resource/content/1/Software%20process%20structure%20and%20model-doc.pdf*
- *https://www.informit.com/articles/article.aspx?p=24671&seqNum=7*
- *https://www.ques10.com/p/8333/what-is-agility-in-context-of-software-engineeri-1/*
- *Roger S.,Pressman,ed.(2010)Software Engineering: A Practitioner’s Approach.McGraw-Hill Companies.*

❄❄❄❄❄❄❄

2

REQUIREMENT ANALYSIS AND SYSTEM MODELING

Unit Structure :

2.0 OBJECTIVES

After going through this chapter, you will be able to:

- Requirement Engineering
- SRS
- UML in software Engineering

2.1 INTRODUCTION

The intent of requirements engineering is to provide all parties with a written understanding of the problem. This can be achieved through a number of work products: usage scenarios, functions and features lists, requirements models, or a specification.

2.2 REQUIREMENTS ENGINEERING

Requirements engineering refers to the wide range of jobs and methods that help one understand requirements. Requirements engineering is a significant software engineering activity that starts during the communication activity and continues through the modeling activity from the standpoint of the software process. It needs to be modified to meet the requirements of the work being done, the project, the product, and the process.It encompasses seven distinct tasks: inception, elicitation, elaboration, negotiation, specification, validation, and management.

Inception:

Stakeholders from the business community (e.g., business managers, marketing people, product managers) define a business case for the idea, try to identify the breadth and depth of the market, do a rough feasibility analysis, and identify a working description of the project's scope.

Elicitation.:

Ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis.

Elaboration:

The information obtained from the customer during inception and elicitation is expanded and refined during elaboration.Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actor)will interact with the system.

Negotiation:

It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources.These conflicts can be reconciled through a process of negotiation.Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority. Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

Specification:

In the context of computer-based systems (and software), the term specification means different things to different people. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

Validation: The work products produced as a consequence of requirements engineering are assessed for quality during a validation step. Requirements validation examines the specification to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

Requirements management:

Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds.

functions and features lists, requirements models, or a specification.

2.3 ELICITING REQUIREMENTS

Requirements elicitation (also called requirements gathering) combines elements of problem solving, elaboration, negotiation, and specification.

1. Collaborative requirements gathering

- Gathering the requirements by conducting the meetings between developer and customer.
- Fix the rules for preparation and participation.
- The main motive is to identify the problem, give the solutions for the elements, negotiate the different approaches and specify the primary set of solution requirements in an environment which is valuable for achieving the goal.

2. Quality Function Deployment (QFD)

- In this technique, translate the customer need into the technical requirement for the software.
- The QFD system designs software according to the demands of the customer.

QFD consist of three types of requirement:

Normal requirements

- The objective and goal are stated for the system through the meetings with the customer.
- For customer satisfaction these requirements should be there.

Expected requirement

- These requirements are implicit.
- These are the basic requirements that are not clearly told by the customer, but also the customer expects that requirement.

Exciting requirements

- These features are beyond the expectation of the customer.
- The developer adds some additional features or unexpected features into the software to make the customer more satisfied. **For example,** the mobile phone with standard features, but the developer adds few additional functionalities like voice searching, multi-touch screen etc. then the customer is more excited about that feature.

3. Usage scenarios

- Until the software team does not understand how the features and function are used by the end users it is difficult to move technical activities.
- To achieve the above problem the software team produces a set of structures that identify the usage for the software.
- This structure is called 'Use Cases'.

4. Elicitation work product

- The work product created as a result of requirement elicitation that is depending on the size of the system or product to be built.
- The work product consists of a statement need, feasibility, statement scope for the system.
- It also consists of a list of users participate in the requirement elicitation.

2.4 SRS VALIDATION

A software requirements specification (SRS) is a document that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.The work products created as a result of requirements engineering are checked for consistency, omissions, and ambiguity during the validation process. The main goal is to guarantee that the SRS correctly and clearly represents the real needs.

Requirements validation is similar to requirements analysis as both processes review the gathered requirements. Requirements validation studies the ‘final draft’ of the requirements document while requirements analysis studies the ‘raw requirements’ from the system stakeholders (users). Requirements validation and requirements analysis can be summarized as follows:

Requirements validation: Have we got the requirements right?

Requirements analysis: Have we got the right requirements?

Requirements validation determines whether the requirements are substantial to design the system. The problems encountered during requirements validation are listed below.

- Unclear stated requirements
- Conflicting requirements are not detected during requirements analysis
- Errors in the requirements elicitation and analysis
- Lack of conformance to quality standards.

To avoid the problems stated above, a **requirements review** is conducted, which consists of a review team that performs a systematic analysis of the requirements.

Requirements Validation Checklist

It is often useful to examine each requirement against a set of checklist questions

- Are requirements stated clearly? Can they be misinterpreted?
- Is the source (e.g., a person, a regulation, a document) of the requirement identified? Has the final statement of the requirement been examined by or against the original source?
- Is the requirement bounded in quantitative terms?
- What other requirements relate to this requirement? Are they clearly noted via a cross-reference matrix or other mechanism?
- Does the requirement violate any system domain constraints?
- Is the requirement testable? If so, can we specify tests (sometimes called validation criteria) to exercise the requirement?
- Is the requirement traceable to any system model that has been created?
- Is the requirement traceable to overall system/product objectives?
- Is the specification structured in a way that leads to easy understanding, easy reference, and easy translation into more technical work products?
- Has an index for the specification been created?
- Have requirements associated with performance, behavior, and operational characteristics been clearly stated? What requirements appear to be implicit?

2.4 COMPONENTS OF SRS

SRS should have these components

1. **Functional Requirements**

   Functional requirements specify what output should be produced from the given inputs. So they basically describe the connectivity between the input and output of the system. For each functional requirement:

   1. A detailed description of all the data inputs and their sources, the units of measure, and the range of valid inputs be specified:
   2. All the operations to be performed on the input data obtain the output should be specified, and
   3. Care must be taken not to specify any algorithms that are not parts of the system but that may be needed to implement the system.
   4. It must clearly state what the system should do if system behaves abnormally when any invalid input is given or due to some error during computation. Specifically, it should specify the behavior of the system for invalid inputs and invalid outputs.

2. **Performance Requirements (Speed Requirements)**

   This part of an SRS specifies the performance constraints on the software system. All the requirements related to the performance characteristics of the system must be clearly specified. Performance requirements are typically expressed as processed transactions per second or response time from the system for a user event or screen refresh time or a combination of these. It is a good idea to pin down performance requirements for the most used or critical transactions, user events and screens.

3. **Design Constraints**

   The client environment may restrict the designer to include some design constraints that must be followed. The various design constraints are standard compliance, resource limits, operating environment, reliability and security requirements and policies that may have an impact on the design of the system. An SRS should identify and specify all such constraints.

   Standard Compliance: It specifies the requirements for the standard the system must follow. The standards may include the report format and according procedures.

   Hardware Limitations: The software needs some existing or predetermined hardware to operate, thus imposing restrictions on the design. Hardware limitations can include the types of machines to be used, operating system availability, memory space etc.

Fault Tolerance: Fault tolerance requirements can place a major constraint on how the system is to be designed. Fault tolerance requirements often make the system more complex and expensive, so they should be minimized.

Security: Currently security requirements have become essential and major for all types of systems. Security requirements place restrictions on the use of certain commands, control access to databases, provide different kinds of access, requirements for different people, require the use of passwords and cryptography techniques, and maintain a log of activities in the system.

4. External Interface Requirements

For each external interface requirements:

1. All the possible interactions of the software with people hardware and other software should be clearly specified,
2. The characteristics of each user interface of the software product should be specified and
3. The SRS should specify the logical characteristics of each interface between the software product and the hardware components for hardware interfacing.

2.5 CHARACTERISTICS OF SRS

Following are the Characteristics of a good SRS document:

1. **Correctness:** User review is used to provide the accuracy of requirements stated in the SRS. SRS is said to be perfect if it covers all the needs that are truly expected from the system.
2. **Complete**: software system will perform each and every function as per the SRS.A SRS is complete if everything the software is supposed to do and the responses of the software to all classes of input data are specified in SRS.To ensure completeness, one has to detect the absence of specification which is much harder to determine.
3. **Consistency:** Requirements at all levels must be consistent with each other .any conflict between requirements within the SRS must be identified and resolved. The types of conflicts that generally occur are: For example, The format of an output report may be described in one requirement as tabular but in another as textual.
4. **Clarity**: The documented requirement should lead to only a single interpretation, independent of the person or the time when the interpretation is done. The SRS needs to be unambiguous to the authors, the users, other reviewers as well as the developers and testers who will use the document. So SRS writers should be careful about ambiguity.

5. **Ranking :** Generally, the requirements stated according to their priorities are critical, others are important but not critical, and there are some which are desirable but not very important.

6. **Modifiability:** SRS should be made as modifiable as likely and should be capable of quickly obtaining changes to the system to some extent.

7. **Traceability:** The SRS is traceable if the origin of each of the requirements is clear and if it facilitates the referencing of each condition in future development or enhancement documentation.

There are two types of Traceability:

1. **Backward Traceability:** This depends upon each requirement explicitly referencing its source in earlier documents.

2. **Forward Traceability:** This depends upon each element in the SRS having a unique name or reference number.The forward traceability of the SRS is especially crucial when the software product enters the operation and maintenance phase. As code and design documents are modified, it is necessary to be able to ascertain the complete set of requirements that may be concerned by those modifications.

3. **Testability:** An SRS should be written in such a method that it is simple to generate test cases and test plans from the report.

2.6 OBJECT-ORIENTED DESIGN USING THE UML

Object Oriented Design (OOD) is the process of defining the objects and their interactions to solve a problem that was identified and documented during the Object Oriented Analysis (OOA). OOD is a design model that is considered as a blueprint for software construction.

The general steps that a software engineer should take to execute object-oriented design are as follows:

1. Identify each subsystem and assign responsibilities to it.
2. Select a design approach for putting task management, interface support, and data management into practice.
3. Create a system-appropriate control mechanism.
4. Create procedural representations for each action and data structures for class attributes to do object design.
5. Perform message design using collaborations between objects and object relationships.
6. Create the messaging model
7. Examine the design model and iterate as necessary

UML

The Unified Modelling Language, or the UML, is a graphical modeling language that provides us with a syntax for describing the major elements (called artifacts in the UML) of software systems.UML has a lot of different diagrams (models). The reason for this is that it is possible to look at a system from different viewpoints. UML being a graphical language includes nine such diagram models):

- Class diagram
- Object diagram
- Use case diagram
- Sequence diagram
- Collaboration diagram
- Statechart diagram
- Activity diagram
- Component diagram
- Deployment diagram

2.6.1 Class Diagram

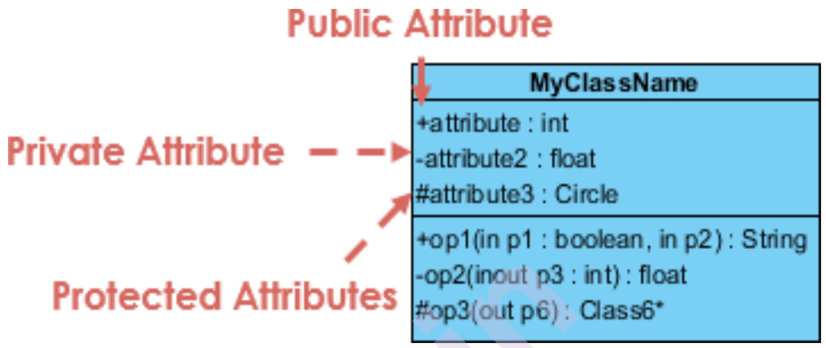
In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.A class represent a concept which encapsulates state (attributes) and behavior (operations). Each attribute has a type. Each operation has a signature. *The class name is the only mandatory information*.

Class Notation

A class notation consists of three parts:

1. **Class Name**
   - The name of the class appears in the first partition.
2. **Class Attributes**
   - Attributes are shown in the second partition.
   - The attribute type is shown after the colon.
   - Attributes map onto member variables (data members) in code.
3. **Class Operations** (Methods)
   - Operations are shown in the third partition. They are services the class provides.

- The return type of a method is shown after the colon at the end of the method signature.
- The return type of method parameters is shown after the colon following the parameter name.
- Operations map onto class methods in code
- The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.



2.6.2 Object Diagram

Object is an instance of a class in a particular moment in runtime that can have its own state and data values.Before creating a class diagram, their might need to create an object diagram to discover facts about specific model elements and their links.They are useful to explain smaller portions of your system, when your system class diagram is very complex, and also sometimes modeling recursive relationship in diagram.

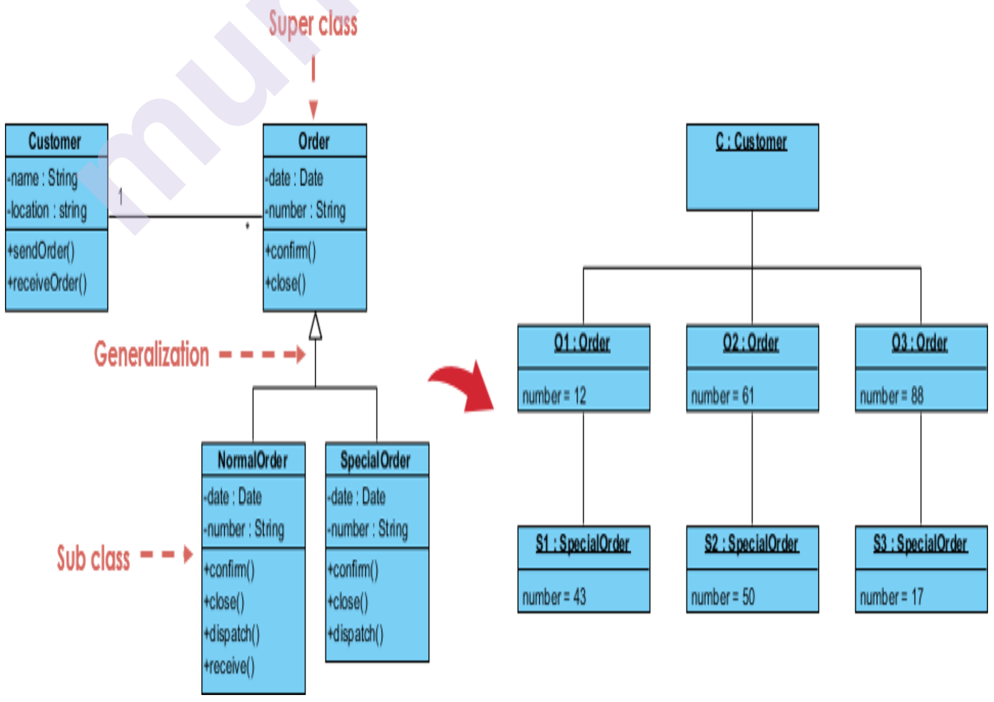


Fig 2.6.2 Class Diagram

Object Diagram

- Every object is actually symbolized like a rectangle, that offers the name from the object and its class underlined as well as divided with a colon.
- Similar to classes, a list of object attributes inside a separate compartment can be listed. However, unlike classes, object attributes should have values assigned for them.

2.6.3. Use Case Diagram

- A use case diagram can summarize the details of your system's users (also known as actors) and their interactions with the system.
- use case diagrams are ideal for:
  - Representing the goals of system-user interactions
  - Defining and organizing functional requirements in a system
  - Specifying the context and requirements of a system
  - Modeling the basic flow of events in a use case

The Common components of use case diagram include:

- Actors: The users that interact with a system. An actor can be a person, an organization, or an outside system that interacts with your application or system. They must be external objects that produce or consume data.
- System: A specific sequence of actions and interactions between actors and the system. A system may also be referred to as a scenario.
- Goals: The end result of most use cases. A successful diagram should describe the activities and variants used to reach the goal.

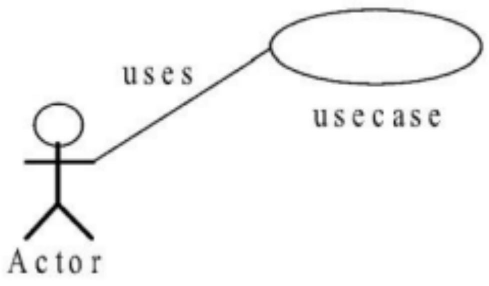


Figure : Basic Notation of a Use Case Diagram

2.6.4 Sequence Diagram

The sequence diagram is used primarily to show the interactions between objects in the sequential order that those interactions occur. One of the primary uses of sequence diagrams is in the transition from requirements expressed as use cases to the next and more formal level of refinement.Sequence Diagrams are driven by the Use Cases which are the system requirements. In this form objects are shown as vertical lines with the messages as horizontal lines between them. The sequence of messages is indicated by reading down the page (read left to right and descending). Sequence Diagrams are about deciding and modeling "how" the system will achieve "what" we described in the Use Case model.

Example of a "Make a Cup of Tea" sequence diagram generated from its corresponding use case description is as shown in Figure below.

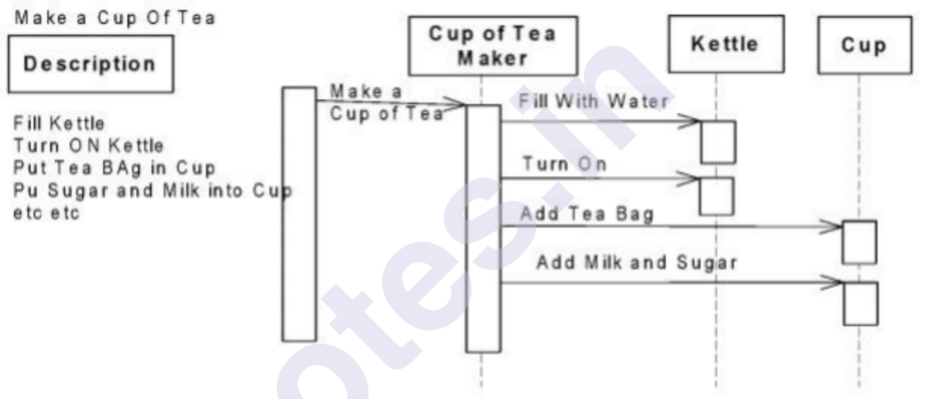


Figure: A Sequence Diagram for "Make a Cup of Tea" Use Case

2.6.5. Collaboration diagram

Communication diagrams, formerly known as collaboration diagrams, are almost identical to sequence diagrams in UML, but they focus more on the relationships of objects—how they associate and connect through messages in a sequence rather than interactions.A communication diagram offers the same information as a sequence diagram, but while a sequence diagram emphasizes the time and order of events, a communication diagram emphasizes the messages exchanged between objects in an application.

<u>**Symbols and notations of communication diagrams**</u>

- Rectangles represent objects that make up the application.
- Lines between class instances represent the relationships between different parts of the application.
- Arrows represent the messages that are sent between objects.
- Numbering lets you know in what order the messages are sent and how many messages are required to finish a process.

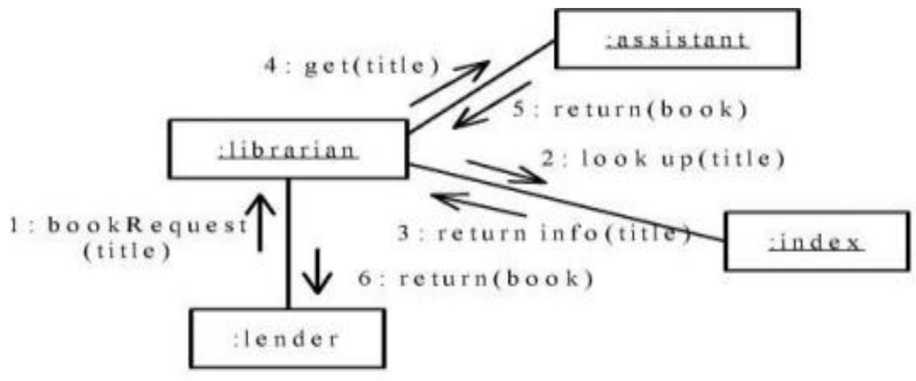
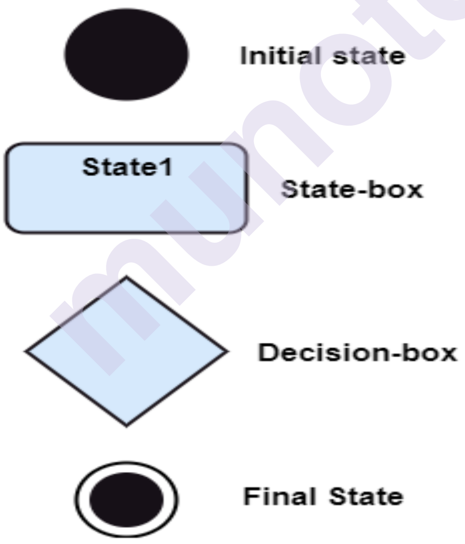


Figure: Example of a Collaboration Diagram

2.6.6 State Chart Diagram

A state diagram depicts how classes behave in response to external inputs. A state diagram, in particular, illustrates the behavior of a single item in response to a series of events in a system. It is sometimes referred to as a Harel state chart or a state machine diagram. This UML diagram depicts the dynamic flow of control from one state to the next of a specific item inside a system.

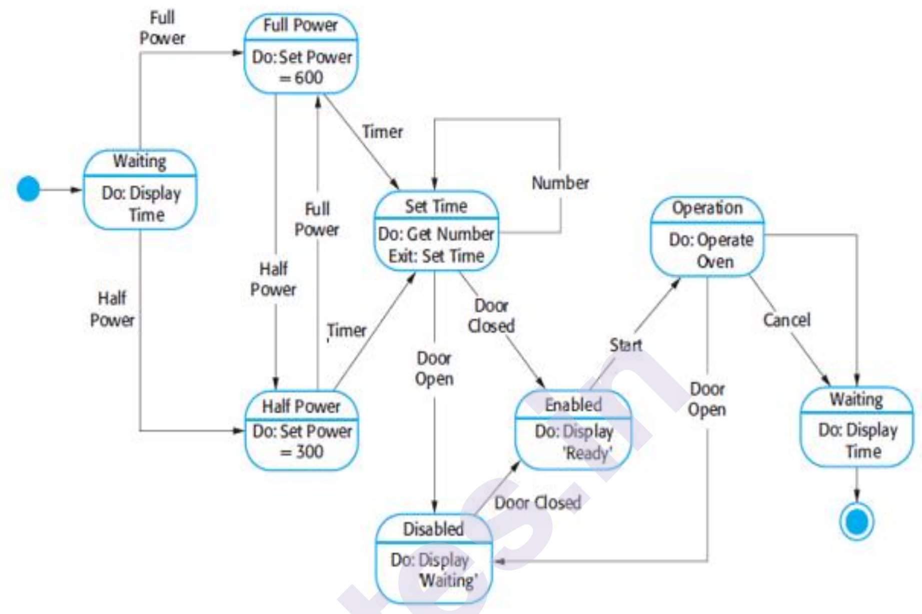
<u>Notation of a State Machine Diagram</u>



- **State**:States represent situations during the life of an object.
- **Transition:**A solid arrow represents the path between different states of an object. Label the transition with the event that triggered it and the action that results from it. A state can have a transition that points back to itself.
- **Decision**:It is of diamond shape that represents the decisions to be made on the basis of an evaluated guard.

- **Initial State:**A filled circle followed by an arrow represents the object's initial state.
- **Final State:**An arrow pointing to a filled circle nested inside another circle represents the object's final state.

Fig 2.6.6 State Chart Diagram



2.6.7 Activity Diagram

A UML activity diagram depicts the dynamic behavior of a system or part of a system through the flow of control between actions that the system performs. It is similar to a flowchart except that an activity diagram can show concurrent flows.

- The main component of an activity diagram is an action node, represented by a **rounded rectangle.**
- **Arrows** from one action node to another indicate the flow of control.
- A **solid black dot** forms the initial node that indicates the starting point of the activity.
- A **black dot surrounded by a black circle** is the final node indicating the end of the activity.
- A **fork** represents the separation of activities into two or more concurrent activities. It is drawn as a **horizontal black bar** with one arrow pointing to it and two or more arrows pointing out from it

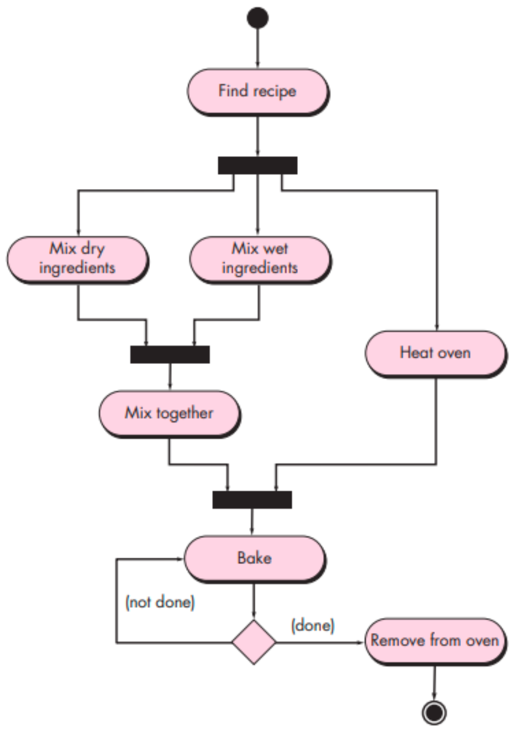
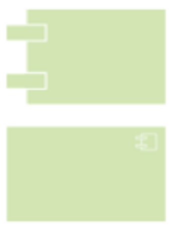


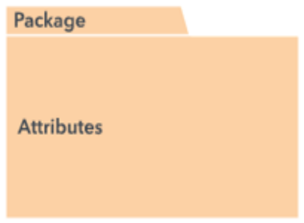
Fig 2.6.7 Activity Diagram

2.6.8 Component Diagram

The purpose of a component diagram is to show the relationship between different components in a system.The term "component" refers to a module of classes that represent independent systems or subsystems with the ability to interface with the rest of the system.A component diagram is similar to the package diagram. It works in the same way as the package diagram, showing the organizations and dependencies among a set of components. Component diagrams address the static implementation view of a system. Component diagrams emphasize the physical software entity e.g. files headers, executables, link-libraries etc, rather than the logical partitioning of the package diagram. It is based heavily on the package diagram, but has added ".dll" to handle I/O, and has added a test harness executable. Not heavily used, but can be 188 helpful in mapping the physical, real life software code and dependencies between them. Figure 2.8 shows a symbol used for a software component.


Component Diagram

Node Symbol



Package Symbol

2.6.9 Deployment Diagram

In UML, deployment diagrams model the physical architecture of a system. Deployment diagrams show the relationships between the software and hardware components in the system and the physical distribution of the processing.Deployment diagrams, which you typically prepare during the implementation phase of development, show the physical arrangement of the nodes in a distributed system, the artifacts that are stored on each node, and the components and other elements that the artifacts implement. Nodes represent hardware devices such as computers, sensors, and printers, as well as other devices that support the runtime environment of a system. Communication paths and deploy relationships model the connections in the system.

Deployment diagrams are effective for visualizing, specifying, and documenting the following types of systems:

- Embedded systems that use hardware that is controlled by external stimuli; for example, a display that is controlled by temperature change
- Client/server systems that typically distinguish between the user interface and the persistent data of a system
- Distributed systems that have multiple servers and can host multiple versions of software artifacts, some of which might even migrate from node to node.

LET US SUM UP

This chapter provides understanding about requirement engineering.It covers about SRS and its validation .The Second half of the chapter focuses on different diagrams available as part of UML which provide a rich set of representational forms for the design model.

QUESTIONS

1. Explain different tasks in Requirement Engineering.
2. Describe Eliciting requirements.
3. Explain SRS and its Validation
4. What are the components of SRS?Explain it
5. What are the characteristics of SRS
6. Explain Class diagram
7. Explain Object diagram
8. Explain Use case diagram

9. Explain Sequence diagram
10. Explain Collaboration diagram
11. Explain Statechart diagram
12. Explain Activity diagram
13. Explain Component diagram
14. Explain Deployment diagram

REFERENCES:

- *https://www.tutorialride.com/software-engineering/software-requirements-engineering.htm#:~:text=Collaborative%20requirements%20gathering&text=Fix%20the%20rules%20for%20preparation,is%20valuable%20for%20achieving%20goal.*
- *https://www.tutorsglobe.com/homework-help/software-engineering/components-of-the-srs-7746.aspx*
- *https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/*
- *https://developer.ibm.com/articles/the-sequence-diagram/*
- *https://www.lucidchart.com/pages/uml-sequence-diagram*
- *https://www.lucidchart.com/pages/uml-communication-diagram*

❄❄❄❄❄❄❄

3

SYSTEM DESIGN

Unit Structure :

3.0 OBJECTIVES

The objectives of this chapter consist of:

- Understanding the open-closed principle, connection, coherence, and modularity design principles
- Getting familiar with function-oriented system's structure using the structure chart notation and the structured design process used to create the system's structure chart
- Fundamentals and process of object-orientation design for a system
- Understanding the guidelines for creating thorough designs, methods for confirming designs, and measures for measuring design complexity
- comprehend the significance of software architecture
- understand the choices that need to be taken during the architectural design process regarding the system architecture
- to learned about architectural patterns, tried-and-true methods of structuring system architectures that can be used to system designs

3.1 INTRODUCTION

When the architecture and document illustrating project has been created that needs to produced is presented, the design process may start. We further enhance the architecture during design. Design is typically concentrated on what we have dubbed the module perspective. That is, we decide which modules need to be developed and which ones the system should have during the design phase. Often, the module view can be thought of as the architecture's individual components organized into modules. Here, the framework establishes component's development architecture. This straightforward component to module mapping, nevertheless, might not always be accurate. In that instance, it is imperative that we make sure the module view we produced during design adheres to the architecture.

A system's design is simply a blueprint or strategy for resolving a problem with the system. Here, a system is viewed as a collection of elements that communicate with one another to provide a particular behaviour or set of services for its environment.

There are typically two levels to the architecture development process. Choosing required elements, their parameters, and how they should be connected are the main concerns at the initial level. This is what is referred to as the high-level or module design. The intramural module architecture and how the requirements can be satisfied is decided at the second stage. To make the system design sufficiently complete for coding, framework incorporated detailing architecture. A methodology is a methodical process that involves using a set of techniques and principles to create a design. While most design approaches concentrate on architecture, they don't boil down the process to set of instructions the designer may follow without thinking.

3.2 SYSTEM/SOFTWARE DESIGN

If a system constructed exactly in accordance with the specification meet needs of that model, the architecture of that system is correct. Producing accurate designs is undoubtedly the aim of the design process. There can be several accurate designs, thus accuracy is not the only factor considered throughout the design phase. Not just creating a model architecture is the objective. Instead, it aims at coming up with the greatest design you can while staying within the constraints set through specifications.

We must define some evaluation criteria before we can assess a design. We shall concentrate on a system's modularity, determined by architecture, as primary requirement for analysis. It is obvious that modularity is a desired quality. System debugging is facilitated by modularity because it makes it simpler to isolate a system issue to a specific module. Modularity also facilitates system repair since replacing a component of the system only impacts a small number of other components.

Simply dividing a software system into a number of modules won't make it modular. Each module must have a clearly illustrated set of rules and specifications & a transparent alliance for communication with different entities in order to be considered modular. Two modularization criteria that are frequently employed together are coupling and cohesion.

3.3 ARCHITECTURAL DESIGN

Understanding how a system should be structured and creating that system's general structure are both aspects of architectural design. The architectural design level of the s/w architecture process is first one in the system evolution procedure. It determines primary entities of a model& their connections, requirements engineering serves as the important connectivity between architecture and engineering needs.

It is widely acknowledged that in agile processes, the establishment of a generic system architecture should take place early in the development process. Architectural progress that takes place incrementally rarely works. Restructuring design is costly, whereas refactoring components in reaction to changes typically rather simple.

To understand the concept, consider about Figure 1. The components that need to be built are shown in an abstract model of the architecture for a packing robot system. This robotic system is capable of packing a variety of objects. It picks out items on a conveyor, determines the kind of item, and chooses the appropriate packing all using a vision component. After that, the system transports items off of the delivery belt for packaging. It loads packaged goods onto a different conveyor. These elements and the connections between them are displayed in the architectural model.

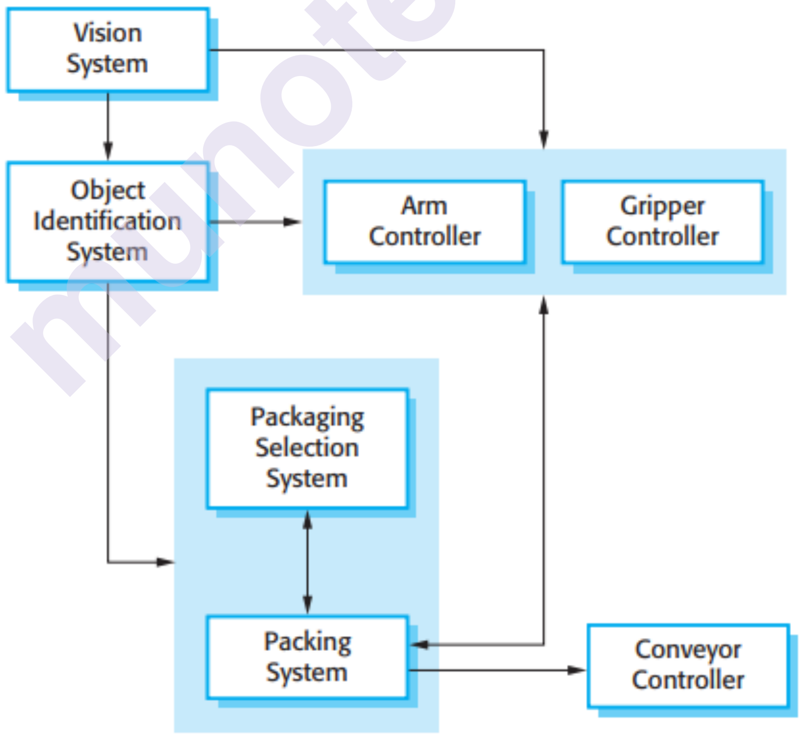


Figure 1: The architecture of a packing robot control system

Architectural design and requirements engineering procedures frequently overlap one another in practise. A system specification should ideally not contain any design data. With the exception of very small systems, this is unrealistic. The specification must typically be organised and structured using architectural decomposition. As a result, you might suggest an abstract system architecture where link collections of system features or functions to substantial parts or subsystems is established. The system's requirements and features can then be discussed with stakeholders using this breakdown.

There are two degrees of abstraction available when designing software architectures:

1] Small-scale architecture focusses on design of specific projects. Here interested lies in how a programme is broken down into its component parts.

2] Architecture, in its broadest sense, refers to the design of intricate module incorporating different sub modules, programmes & programme elements. This corporate model dispersed across numerous computers, some of which may be owned and operated by various businesses.

3.4 COUPLING

If one module can run entirely on its own without the other, they are said to be independent. It goes without saying that if two modules are independent, they can be solved and modified independently. The modules of a system must interact with one another in order to achieve the desired outward behaviour of the system, hence they cannot all be independent of one another. The more linkages there are between modules, the more interdependent they are in that it takes a deeper understanding of one module to comprehend or address the problem in the other. Therefore, it is simpler to understand one module without understanding the other links between them. The concept of coupling makes an attempt to express this idea of "how strongly" certain modules are related.

The degree of linkages or the degree of interdependence between modules is referred to as coupling. In general, A and B are more tightly related the more information we need to grasp module A before we can fully comprehend module B. While "loosely coupled" have frail connectivity, "highly coupled" are connected by strong interconnections. There are no linkages between independent components. The s/w modules are developed during architectural formulation, therefore connection across them is primarily determined at that time and cannot be lessened during implementation.

The more complicated and obscure the interface between modules, the more coupling there is. The number of alliances should be kept to a minimum in order to maintain low coupling. Information is passed to and from other modules through a module's interface.

Another element affecting coupling is interface complexity. The degree of coupling will be larger the more intricate each interaction is. For instance, both the quantity and complexity of the items supplied as parameters affect how complicated the entry interface for a procedure is. Interface complexity must support the necessary communication between modules to some extent. However, this minimum is frequently exceeded. We are unnecessarily increasing the coupling by breaking the record. Basically, we should keep a module's interface as straightforward and condensed as possible.

The third key element influencing coupling is the data workflow process at the alliances. Data and control are the two types of information that can travel via an interface. It is more challenging to comprehend & offer when control is passed or received. A module transfers data information when it provides some data as input & receives some as o/p. Table 1 summarises how these three elements affect coupling.

Table 1: Parameters influencing coupling

	Interface Complexity	**Type of Connection**	**Type of Communication**
Low	Simple obvious	To module by name	Data
			Control
High	Complicated obscure	To internal elements	Hybrid

Due to fact that objects have a greater semantic richness than functions, coupling takes on a slightly different appearance in OO systems. They are of three types:

– Interaction

– Component

– Inheritance

Methods of one class calling methods of another cause's interaction coupling. This circumstance resembles a function calling another function in many respects, and as a result, this coupling resembles coupling between functional modules. Within this group, coupling is smaller when only data is sent, but it increases when data flows since the invoked type affects how calling function is executed. Additionally, coupling increases as data transmission volume rises.

When two classes communicate and one class contains variables from the other class, this is referred to as component coupling. There are three distinct circumstances in which this may occur. If a class C has an instance variable of type C1, a method with a parameter of type C1, a method with a local variable of type C1, or all three, the class C can be a component connected with another class C1. Because any object from any subclass may be used at runtime, when C is component coupled with C1, it has the potential to be component coupled with all subclasses of C1. It should be obvious that there will almost always be interaction coupling whenever there is component coupling. If the variables of class C1 are either in the signatures of the methods of class C or in some properties of class C, component coupling is thought to be weakest (and hence most desirable). If there is interaction through local variables, this interaction is not apparent from the outside, increasing coupling.

3.5 COHESION

This indicates how closely related its internal components are to one another. The cohesion of a module informs the designer of whether the various components of a module should be placed together in the same module. Coupling and cohesion are connected. Typically, there is less coupling between modules the more cohesive each module is within the system. Although this association is not exact, it has been seen in real-world situations. Different levels of cohesiveness exist: Coincidental, logical, temporal, procedural, communicational, sequential and functional.

The levels are coincidental (lowest) and functional (highest). When there is no significant relationship between the components of a module, there is accidental cohesiveness. An existing programme can be "modularized" by breaking it up into smaller bits and turning each of those pieces into a separate module. A module is likely to exhibit coincidental coherence if it combines a portion of code that appears in multiple places in order to reduce duplicate code.

If a module's components perform tasks that belong to the same logical class and there is some logical relationship between them, the module is said to have logical cohesiveness. A module that handles all the inputs or all the outputs is a common illustration of this type of cohesiveness. If we want to enter or output a certain record in such a scenario, we must to communicate this to the module. This is frequently accomplished by including a specific status flag that will be utilised to select which module statements to execute. Such a module typically has complex and awkward code, in addition to creating hybrid information flow between modules, which is typically the worst type of coupling between modules. In general, it's best to stay away from logically cohesive units.

Similar to logical coherence, temporal cohesion refers to the execution of elements that are related in time. Typically, modules that carry out tasks like "initialization," "cleanup," and "termination" are time-bound. The pieces in a temporally bound module are logically related, but because they are all executed simultaneously, temporal cohesiveness is stronger than logical cohesion. By doing so, the issue of passing the flag is avoided, and the code is typically shorter.

A procedurally cohesive module is made up of components from a single procedural unit. A module's loop or series of decision statements, for instance, could be concatenated to create a new module. When a modular structure is derived from a type of flowchart, procedurally coherent modules frequently appear. Functional boundaries are frequently crossed through procedural coherence. Several functions or merely a portion of a function may be present in a module with only procedural cohesiveness.

Elements in a module with communicational cohesion are connected by a reference to the same input or output data. In other words, in a communication-bound module, the components are grouped together because they share input or output data. This might include a module to "print and punch record," for instance. Modules with good communication may serve multiple purposes. However, if alternative structures with stronger cohesion cannot be readily recognised, communicational cohesiveness is high enough to be generally accepted.

Sequential cohesiveness occurs when components are grouped together in a module because the output of one serve as the input to another. Sequential cohesion does not offer any recommendations for how to group elements into modules if the result of one element serves as the input to another.

The strongest coherence is functional cohesion. Every component of a module that is functionally bound is connected to carrying out a single function. We don't just mean mathematical functions when we say "function"; we also include modules that achieve a specific task. Functionally cohesive modules can be seen in actions like "calculate square root" and "sort the array."

Cohesion in object-oriented systems has three aspects:

- Method cohesion

- Class cohesion

- Inheritance cohesion

Cohesion in functional modules is the same as method cohesion. It focuses on the rationale behind grouping the various method's code components together. When all of a method's statements work together to implement a single, clearly defined function, this is referred to as cohesiveness at its highest level.

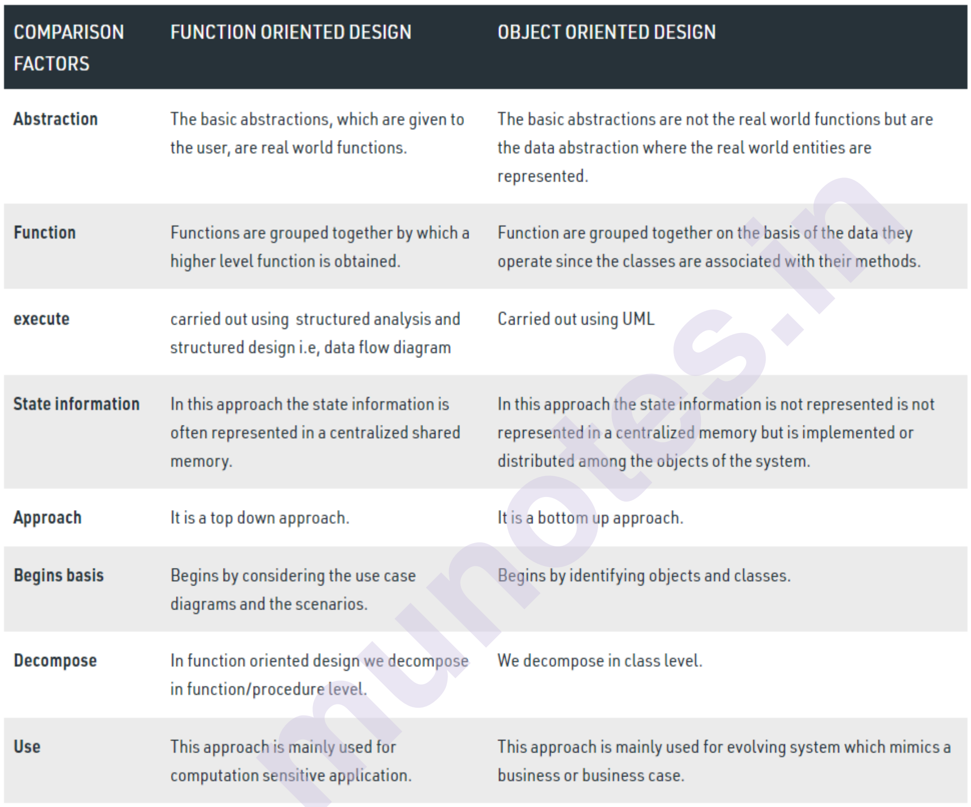
Class cohesiveness examines the rationale behind the grouping of various characteristics and methods in this class. The objective is to have a class that implements a single abstraction or concept, with each component working to support that concept. A designer should attempt to adjust the design such that each class encapsulates a single notion since, generally speaking, anytime many concepts are wrapped within a class, the cohesion of the class is not as high as it could be.

The focus of inheritance cohesion is on the rationale for the grouping of classes in a hierarchy. Inheritance is mostly used to represent generalization-specialization relationships and to reuse code. If the hierarchy encourages generalization-specialization of a certain concept, which is likely to naturally result in code reuse, cohesion is deemed to be good. If the hierarchy's main purpose is code sharing and the superclass and subclass relationships are weak conceptually, it is regarded as lower.

3.6 FUNCTIONAL-ORIENTED VERSUS THE OBJECT-ORIENTED APPROACH

Functional programming uses immutable data to tell the program exactly what to do. Object-oriented programming tells the program how to achieve results through objects altering the program's state. Both paradigms can be used to create elegant code. Table 2 illustrates the detailed comparison between function oriented and object-oriented design approach.

Table 2: Comparison between function oriented and object-oriented design approach



COMPARISON FACTORS	FUNCTION ORIENTED DESIGN	OBJECT ORIENTED DESIGN
Abstraction	The basic abstractions, which are given to the user, are real world functions.	The basic abstractions are not the real world functions but are the data abstraction where the real world entities are represented.
Function	Functions are grouped together by which a higher level function is obtained.	Function are grouped together on the basis of the data they operate since the classes are associated with their methods.
execute	carried out using structured analysis and structured design i.e, data flow diagram	Carried out using UML
State information	In this approach the state information is often represented in a centralized shared memory.	In this approach the state information is not represented is not represented in a centralized memory but is implemented or distributed among the objects of the system.
Approach	It is a top down approach.	It is a bottom up approach.
Begins basis	Begins by considering the use case diagrams and the scenarios.	Begins by identifying objects and classes.
Decompose	In function oriented design we decompose in function/procedure level.	We decompose in class level.
Use	This approach is mainly used for computation sensitive application.	This approach is mainly used for evolving system which mimics a business or business case.

3.7 DESIGN SPECIFICATIONS

The Functional Needs set forth in Design Specifications indicate how a system fulfils those requirements. This may include guidelines for testing particular conditions, configuration options, or a review of functions or code, depending on the system. The functional specification's requirements should all be met.

Design specifications examples:

Good requirements can be tested and are unbiased. Design requirements could consist of

- Data types and specific inputs that must be entered into the system
- Code or calculations used to fulfil specified requirements

- outputs the system produces
- describing the technical safeguards to make systems secure
- Indicate how the system complies with any applicable legal requirements.

The Installation Qualification typically includes tests of the System Requirements and the installation procedure. In the operational qualification, input, processing, output, and security testing are often tested.

There is now some discussion in the industry over who needs to examine the Design Specification due to the highly technical nature of most design papers. The System Owner, System Developer, and Quality Assurance must all examine and accept the Design Specification. Quality Assurance certifies that the document complies with the necessary laws and that all requirements were satisfactorily met, although they are not required to review technical data.

The functional requirements document and the design specification may be integrated, depending on the length and complexity of the programme.

3.8 VERIFICATION FOR DESIGN

Before starting the following phase's activities, the design activity's output needs to be confirmed. If the design is expressed in a formal notation for which analysis tools are available, then it can be checked for internal consistency using tools (for example, the modules used by others are defined, a module's interface is consistent with how others use it, data usage is consistent with declaration, etc.). The design cannot be processed using tools if it is not stated in a formal, executable language, hence alternative methods of verification must be utilised. Design review is the method of verification that is most frequently used.

Design reviews are conducted to make sure that the design meets the specifications and is of high quality. If mistakes are committed during the design phase, they will eventually show up in the code and the finished system. It is desirable if design problems are discovered early, before they reveal themselves in the system, as the cost of fixing faults brought on by design errors rises with the delay in recognising the errors. The goal of design reviews is to find design flaws.

Similar to the inspection process, the system design review process involves a group of people meeting to discuss the design in order to identify any flaws or undesired characteristics. A member of the system design team, a member of the detailed design team, the author of the requirements document, the author in charge of maintaining the design document, and an independent software quality engineer are all required to be on the review group. As with any review, it is important to remember that the meeting's goal is to identify design flaws rather than attempt to remedy them; fixing is done later.

Only the designer's imagination can constrain how many ways faults might creep into a design. The fact that the design does not fully fulfil some requirements, however, is the most significant design flaw. For instance, a scenario for an exception situation cannot be handled or a design constraint has not been met. Modularity is the primary factor in determining design excellence. Efficiency is another important factor for which a design is assessed, though, as it is necessary to test whether it can meet performance requirements.

3.9 MONITORING AND CONTROL FOR DESIGN

Control is a management function that aids in error detection and the implementation of corrective measures. This is done to ensure that the organization's stated goals are fulfilled in the desired manner and to reduce deviation from standards.

Monitoring is the process of routinely observing and documenting the actions occurring within a project or programme. It is a procedure for regularly compiling data on every facet of the project.

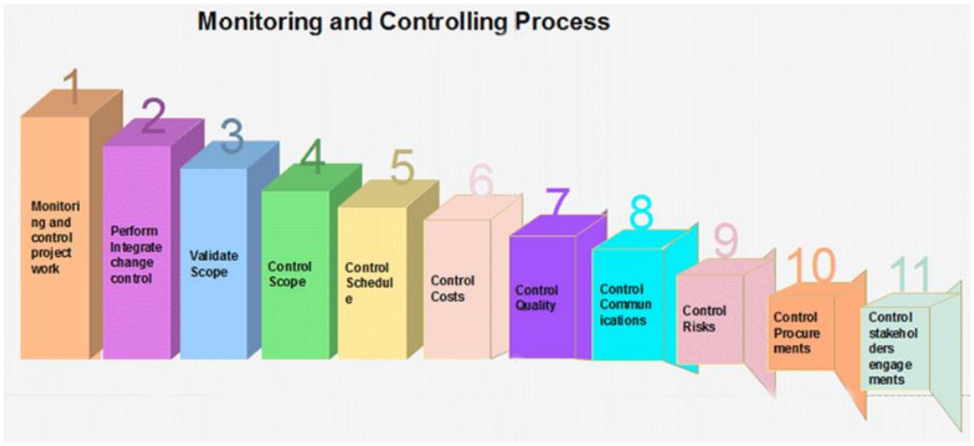
Project control includes the tools, process, people skills and experience, when integrated provide the right information at the right time to enable the right decision to be made. It mainly focusses on attributes such as Why, What, When, Where, and How.

Project monitoring helps to track project performance and progression using key performance indicators (KPIs) agreed during project planning.

- **Project monitoring and control**

  The methods of tracking, reviewing, and regulating the project's performance are known as monitoring and controlling. It also detects any places where adjustments to the project management methodology are necessary and starts making those adjustments.

  Eleven processes make up the Monitoring & Controlling process group, and they are as follows:



1] Monitor and control project work: The first stage, known as "monitor and control project work," is the umbrella step for all further monitoring and regulating operations.

2] Perform integrated change control: The procedures necessary to modify the project plan. The program is modified and reapproved by the project sponsor if adjustments to the schedule, budget, or any other aspect of the project management plan are required.

3] Validate scope: The procedures necessary to obtain project deliverable approval.

4] Control scope: Making sure that the project's scope does not change and that no unapproved actions are taken in accordance with the plan (scope creep).

5] Control schedule: The activities involved in making sure that project work is carried out in accordance with the schedule and that project deadlines are reached.

6] Control costs: The activities necessary to guarantee that the project expenses adhere to the approved budget.

7] Control quality: Assuring the project deliverables meet the quality standards outlined in the project management strategy.

8] Control communications: Attending to each project stakeholder's communication demands.

9] Control risks: preventing unforeseen occurrences that could have a detrimental influence on the project's budget, schedule, stakeholder requirements, or any other criterion for project success.

10] Control procurements: Ensuring that the project's vendors and subcontractors achieve the project's objectives.

11] Control stakeholder engagement: The activities necessary to guarantee that each project's stakeholders are happy with the outcome.

SUMMARY

A system's design is a strategy for a course of action that, if carried out, will satisfy the system's requirements and maintain its architectural integrity. The detailed design explains the processing logic of modules, whereas the module-level design identifies the modules that must exist in the system to execute the architecture.

If each module in a system has a clear abstraction and changes to one module have little effect on other modules, the system is said to be modular. Cohesion and coupling are two factors that are taken into account while assessing a design's modularity. Cohesion is a measurement of the degree to which the various components of a module are connected, whereas coupling depicts how dependent modules are on one another. In a design, coupling should typically be reduced and cohesiveness should be increased. The open-closed principle, which states that modules should be available for extension but closed for alteration, should also be supported by the design.

According to the structured design technique, a design should be created (shown as a structure chart) so that the modules have a low amount of coupling and a high level of cohesiveness. In order to accomplish this, the technique divides the system into a number of subsystems, one for managing each significant input, one for managing each major output, and one for managing each major transformation. This neatly divides the system into sections that each independently address various issues.

LIST OF REFERENCES

1] Software Engineering, A Practitioner's Approach, Roger S, Pressman (2014).

2] Software Engineering, Ian Sommerville, Pearson Education.

3] Fundamentals of Software Engineering, Fourth Edition, Rajib Mall, PHI.

4] Software Engineering: Principles and Practices, Hans Van Vliet, John Wiley & Sons.

5] A Concise Introduction to Software Engineering, Pankaj Jalote, Springer.

UNIT END EXERCISES

1] Explain the concept of system/ software design.

2] Write a note on architectural design.

3] Discuss on coupling.

4] Write a note on cohesion.

5] Illustrate the comparison between function-oriented and object-oriented approach.

6] Write a note on design specification.

7] Explain the concept of verification for design.

8] Define control and monitoring. Explain in brief the eleven steps involved in project monitoring and control process.

❄❄❄❄❄❄❄

4

SOFTWARE MEASUREMENT AND METRICS

Unit Structure :

4.0 OBJECTIVES

- To understand the applicability of metrics in software engineering
- To get familiar with how the real time entities are associated with metrics and measurement
- To understand how metrics are used to evaluate the product's quality
- To get real time indication of effectiveness of test cases

4.1 INTRODUCTION

Measurement is a crucial component of every engineering process. You can evaluate the qualities of the engineered goods or systems you produce using metrics for knowing the characteristics of system you develop. SE however, is not based on fundamental rules of science of matter, unlike other engineering fields. In the domain of software, direct measurements are rare. Metrics and measures are debatable since they are frequently indirect. Fenton speaks to this matter when he says:

Measurement is the process through which the characteristics of real-world entities are given numerical or symbolic values in order to define them in accordance with predetermined guidelines. We can now measure qualities that were once believed to be immeasurable. Even though these were not as accurate as others used to support crucial judgements.

This chapter describes metrics that can be used to evaluate the product's quality as it is being engineered.

4.2 PRODUCT METRICS – MEASURES, METRICS, AND INDICATORS

Perhaps these terms are frequently utilized synonymously, it's crucial to be aware of their little variations. It can be difficult to define measure because it can be employed as a verb or a noun. A measure offers a numerical manifestation of degree, quantity, capability, magnitude of certain feature.

A measurement has been established once a individual datum has been gathered, such as amount of faults. The act of measuring results from the gathering of one or more data points.

A software engineer gathers data and creates metrics in order to produce indicators. An indicator offers perception into the s/w development procedure, a s/w project, or finished result. Indication offers information allowing to improve project, procedure, or system.

4.3 FUNCTION-BASED METRICS

A useful tool for assessing the functionality a system provides is the function point (FP) metric and is used for forecasting following:

(1) the measure or endeavour necessary to describe, write & deploy programme;

(2) quantitative error measurement to discover during deployment

(3) measure of entity to predicted line of codes in the incorporated model.

A quantitative evaluation of software complexity and countable (direct) metrics are used to produce function points. The following definitions apply to information domain values:

The quantity of external inputs (EIs): They delivers unique control driven data and knowledge, whether it comes directly from a user or is sent from another application. It's common practise to reform internal logical files using i/p's (ILFs). It is important to distinguish between inputs and enquiries, that are measured individually.

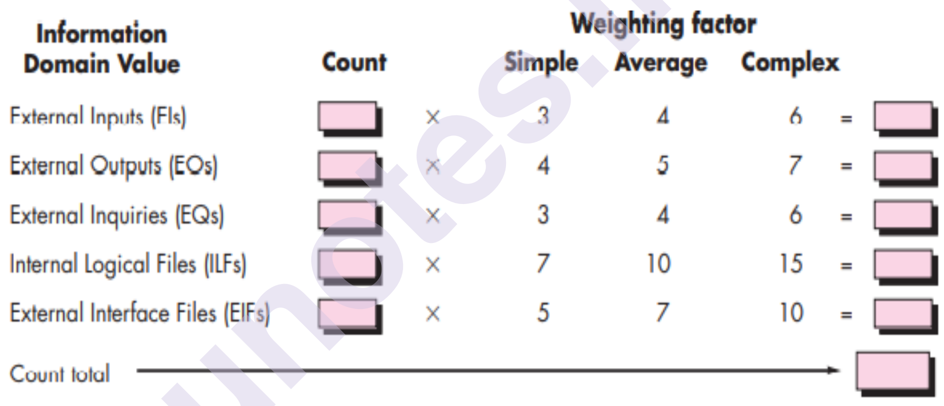
The number of external outputs (EOs): Every EO is procured information from the domain giving the user instruction. External output acknowledges screens, document, fault measurement, etc. A report does not count every piece of data independently.

Number of external enquiries (EQs): It is a web-based i/p triggering an S/W reply through accessible outcome.

Number of internal logical files (ILFs): There are a certain nos. of ILFs, of which is a rational collection of information kept up to date by external inputs and located within the application's boundaries.

Number of external interface files (EIFs): It is a rational collection of information kept apart from the application but contains data that the application may find useful.

Fig 1 is finished & difficulty metric is assigned to every measure once these data have been gathered. Organizations that employ function point methodologies create standards for classifying entries as easy, medium & difficult. Assessment of difficulty is, nevertheless, rather arbitrary.



Information Domain Value	Count		Weighting factor Simple	Average	Complex		
External Inputs (EIs)		×	3	4	6	=	
External Outputs (EOs)		×	4	5	7	=	
External Inquiries (EQs)		×	3	4	6	=	
Internal Logical Files (ILFs)		×	7	10	15	=	
External Interface Files (EIFs)		×	5	7	10	=	
Count total							

Figure 1: Computing function points

To calculate FP below equation is used:

$$FP = \text{Total count} \times [0.65 + 0.01 \times \Sigma (F_i)]$$

Where count total is the summation of all the FP entries that will be obtained from figure 1.

Value adjustment factors (VAF) Fi (i =1 to 14) are determined by the answers to the following inquiries:

1. Is dependable backup and recovery required for the system?
2. Will the application need specialised data communications to send or receive information?
3. Do distributed processing operations exist?
4. Is performance a top priority?

5. Can the system operate in a current, highly trafficked operational environment?
6. Is online data entry required by the system?
7. Do numerous screens or activities need to be included into the input transaction for online data entry?
8. Are the ILFs recovered?
9. How complicated are i/p's, o/p's, binder, or investigations?
10. Is internal processing sophisticated?
11. Was the code created with reuse in mind?
12. Is installation and conversion taken into account in the design?
13. Can the system be installed more than once in various organisations?
14. Is the programme created with the user's ability to alter and use it easily in mind?

Answers to all the above set of questionaries are given on scale from 0 (not useful) to 5 (very useful).

4.4 METRICS FOR OBJECT-ORIENTED DESIGN

As the size and complexity of an OO design model increase, a more impartial assessment of the design's attributes can be advantageous to both the experienced designer and the beginner designer by providing them with information about the design's quality that they would not otherwise have. Whitmire offers nine different and measurably observable properties of an OO design in a thorough examination of S/W:

- Size: Four perspectives - population, volume, length, and functionality are used to define size. A fixed measure or actions, is used to determine population. Volumetric measurements is similar to populace measurements gathered at specific moment. A chain of related design elements can be measured by their length & is used as a metric of length. Functionality measurements offer a hazy picture of the value an OO application brings to the consumer.
- Complexity: Similar to size, there are many different opinions on what constitutes complexity in software. Whitmire analyses the relationships between classes in an OO architecture to understand complexity in terms of structural traits.
- Coupling: In an OO system, this is represented through interlinks across components of the architecture.
- Sufficiency: From perspective of the current application, sufficiency is defined by Whitmire as "extent to which an extraction incorporates the characteristics expected". In other words, we search for: "Which qualities is required to have for me finding it essential?" Fundamentally, a design element is enough if it accurately captures

each the characteristics of modelling process, i.e., if the abstraction (class) has all of the features that are necessary.

- Completeness: "The characteristic set through which we assess the extraction or architectural module" is only distinction between completeness and sufficiency. According to the current use, sufficiency compares the abstraction. Completeness takes into account several viewpoints and poses the following query: "What qualities are necessary to fully represent the issue domain object?" This indirectly implied by the criterion for completeness' consideration of many points of view.
- Cohesion: An OO module must be created in such a way that every computation cooperates to fulfil individual, clear goal, just as its analogue in traditional software. The extent of "the collection of qualities it holds is proportion of difficulty or architecture framework" is used to assess a class's cohesion.
- Primitiveness: This implies along with operations; classes are a quality akin for simplicity. It describes how atomic an operation is, how easily it cannot be created from a series of other actions found within a class. A class with a lot of primitive behaviour only contains primitive operations.
- Similarity: This measure indicates how similar multiple objects are in accordance of their structural formulation and its functionality, behaviour, or requirements.
- Volatility: Architectural modifications might take place when necessities are altered or alterations take place in areas necessary in adaption of the questioned design component. An OO design component's volatility gauges how likely a change is to occur.

4.5 OPERATION-ORIENTED METRIC

By looking at typical characteristics for approaches, some new information can be discovered (operations). According to Lorenz and Kidd, three straightforward measurements are pertinent:

- Average operation size (OS_{avg}): The amount of lines of program forwarded through manipulations can be used to calculate the average operation size (OS_{avg}). It is probably the case that duties have not been fairly distributed within a class when the volume of messages delivered by a single operation increase.
- Operation complexity (OC): Any complexity measure suggested for traditional software can be used to calculate an operation's complexity. The designer should make an effort to keep OC as low as feasible since operations should be restricted to a single task.
- Average number of parameters per operation (NP_{avg}): Complexity of object collaboration increases with the number of operation parameters. NP_{avg} is generally maintained at lower values.

4.6 USER INTERFACE DESIGN METRICS

There is a dearth of data on the metrics that would reveal the interface's usability and quality.

Layout Appropriateness (LA), according to Sears, is valuable development measure for articulation. Conventional GUI aids users in executing tasks by using layout items. Client using a GUI must switched from one format to another in order to complete a job.

According to a research of Web page metrics, the layout's simple qualities can also have a big impact on how well the architectural pattern is received. Amount of text, references, pictures, colours, and typefaces (among other features) on a Web page determine how sophisticated and high-quality the page is regarded to be.

The choice is influenced by measures like LA, but user feedback based on GUI prototypes should be the ultimate arbitrator. According to Nielsen and Levy, "if one selects amongst interface [designs] based simply on users' opinions, one has a relatively large likelihood of success. There is a strong correlation between a user's subjective happiness with a GUI and their average task performance.

4.7 METRICS FOR SOURCE CODE

The first analytic "laws" for computer software were proposed by Halstead's "software science" hypothesis. Halstead used a collection of crude measurements that is determined after program is written or calculated after completion of architectural pattern to create the firmware. These are as follows:

n1: the number of unique operators in a program

n2: the quantity of unique operands in a program

N1: overall operator occurrences

N2: total number of repetitions of the operand

Above fundamental measurements are used by Halstead to create formulations for the complete program.

Halstead shows that length N can be estimated using

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

And program volume is defined as

$$V = N \log_2 (n_1 + n_2)$$

Where V denotes the amount of information (in bits) needed to express a programme, varies depending on the programming language.

Theoretically, a specific algorithm must have a minimal volume. According to Halstead, a volume L is proportion between the vol of a program's dense form and its actual volume. L must actually always be smaller than 1. Primitive measurements allow us to express the volume ratio as

$$L = [(2 / n_1) * (n_2 / N_2)]$$

4.8 HALSTEAD METRICS APPLIED TO TESTING

Utilizing metrics generated from Halstead measurements, testing effort can be calculated. Halstead effort e can be calculated as

$$PL = \frac{1}{(n_1/2) \times (N_2/n_2)}$$

$$e = \frac{V}{PL}$$

The following relationship can be used to estimate the proportion of deployment to be assigned to division k:

$$\text{Percentage of testing effort } (k) = \frac{e(k)}{\Sigma e(i)}$$

where denominator represents total Halstead effort put forth by all of the system's modules.

4.9 METRICS FOR MAINTENANCE

Both the creation & upkeep of latest firmware can make use of all the firmware measures discussed here. However, metrics specifically created for maintenance tasks have been put forth.

Software maturity index (SMI), which is recommended by IEEE Std. 982.1-1988 [IEE93], offers a sign of the solidity of a firmware legacy (required for modifications taking place for every version). The below details are discovered:

M_T = number of modules in the current release
F_c = number of modules in the current release that have been changed
F_a = number of modules in the current release that have been added
F_d = number of modules from the preceding release that were deleted in the current release

The software maturity index is computed in the following manner:

$$\text{SMI} = \frac{M_T - (F_a + F_c + F_d)}{M_T}$$

As SMI gets closer to 1, it starts to sustain. SMI can be utilized as statistic while scheduling firmware monitoring tasks. It is possible to create empirical models for maintenance effort and to associate the average pattern to generate a reveal of a firmware.

4.10 CYCLOMATIC COMPLEXITY

This measure offers numerical assessment of the logical difficulty of a programme. Graph theory serves as the basis for cyclomatic complexity, which offers you a very helpful software metric. One of three methods is used to compute complexity:

1] The number of regions of the flow graphs corresponds to cyclomatic complexity

2] Cyclomatic complexity V(G) for a flow graph G is defined as

$$V(G) = E - N + 2$$

Where,

E: Number of flow graph edges

N: Number of flow graph nodes

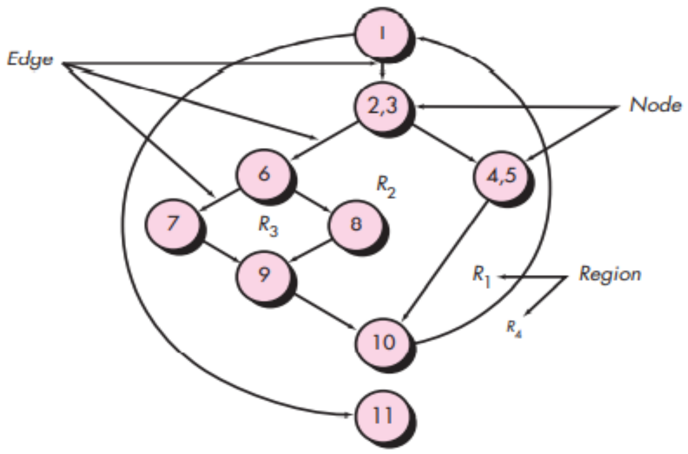
3] Cyclomatic complexity V(G) for a flow graph G is also defined as

$$V(G) = P + 1$$

Where,

P: number of predicate nodes contained in the flow graph G

For example: Consider the flow graph as shown in the following figure



For this figure the cyclomatic complexity can be computed using each of the algorithms just noted

1] The flow graph consists of 4 regions

2] V(G) = 11 edges - 9 nodes + 2 = 4

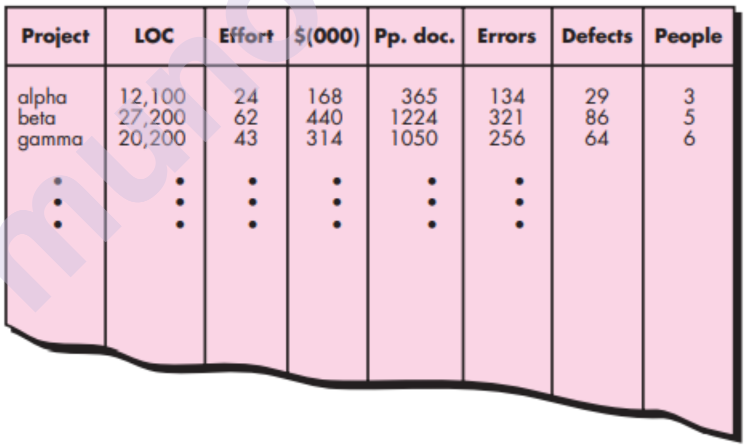
3] V(G) = 3 predicate nodes + 1 = 4

Therefore, the flow graph in Figure has a cyclomatic complexity of 4.

More importantly, the value for V(G) gives you an upper constraint on the number of independent routes that make up the basis set, and thus, an upper bound on the number of tests that must be created and run to ensure that every programme statement is covered.

4.11 SOFTWARE MEASUREMENT: SIZE-ORIENTED METRICS

Size-oriented software metrics are created by averaging productivity and/or quality measurements while taking into account the size of the software that has been created. A table of size-oriented measurements, like the one in Figure 2, can be made if a software organisation keeps simple records. The table includes a list of all finished software development projects over the previous few years together with the relevant project measures. According to the table entry (Figure 2) for project alpha, $168,000 was spent to create 12,100 lines of code over 24 person-months.



Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
⋮	⋮	⋮	⋮	⋮	⋮		

Figure 2: Size-oriented metrics

It should be noted that the work and costs listed in the table cover all software engineering activities, not only coding, including analysis, design, coding, and testing. According to additional data for project alpha, 365 pages of documentation were created, 134 faults were discovered prior to the software's release, and 29 problems were found within the first year of operation after the software's release to the client. The software for project alpha was developed by three persons.

You can decide to use the number of lines of code as a normalisation variable in order to create metrics that can be combined with comparable metrics from other projects. A set of straightforward size-oriented measures may be constructed for each project from the basic data in the table:

- Errors per KLOC (thousand lines of code)
- Defects per KLOC
- $ per KLOC
- Pages of documentation per KLOC

In addition, other interesting metrics can be computed:

- Errors per person-month
- KLOC per person-month
- $ per page of documentation

Not everyone agrees that the best method to evaluate the software process is through size-oriented measures. The usage of lines of code as a crucial indicator is where the majority of the disagreement centres. The LOC measure's proponents assert that LOC is an easily countable "artefact" of all software development projects, that many current software estimation models employ LOC or KLOC as a crucial input, and that a substantial body of research and data based on LOC already exist. Opponents counter that LOC measures are dependent on the programming language being used, that they penalise well-designed but shorter programmes when productivity is taken into account, that they cannot easily accommodate nonprocedural languages, and that their use in estimation necessitates a level of detail that may be challenging to achieve (i.e., the planner must estimate the LOC to be produced long before analysis and design have been completed).

4.12 FUNCTION-ORIENTED METRICS

A measure of the functionality provided by the programme is used as a normalisation value in function-oriented software metrics. The function point is the most used function-oriented measure (FP). The information domain and complexity properties of the software are used to compute the function point.

Like the LOC measure, the function point is debatable. According to supporters, FP is more appealing as an estimation strategy because it is based on data that are more likely to be known early in the lifecycle of a project and is independent of programming language, making it perfect for applications employing traditional and nonprocedural languages. The method's detractors assert that it requires some "sleight of hand" because the computation is based on subjective rather than objective data, that it can be challenging to gather counts of the information domain (and other dimensions) after the fact, and that FP has no immediate physical significance – it's just a number.

4.13 METRICS FOR SOFTWARE QUALITY

Producing a high-quality system, application, or product in a timely manner that answers a market need is the primary objective of software engineering. You must use efficient techniques along with cutting-edge tools inside the framework of an established software process in order to accomplish this goal. If high quality is to be achieved, a competent software engineer (and effective software engineering managers) must measure.

A system, application, or product is only as good as its requirements, which outline the issue, design, which represents the solution, code, which creates an executable program, and tests, which put the software to the test to find bugs. As the software is being developed, measurement can be used to evaluate the quality of the test cases, source code, and requirements and design models that have been produced. You use product metrics to evaluate the quality of the work products produced by software engineering in order to complete this real-time review in an objective rather than subjective manner.

As the project advances, a project manager must also assess quality. Software engineers' private measurements are pooled to produce outcomes at the project level. The main focus at the project level is to measure mistakes and defects, despite the fact that various quality measurements can be gathered. Metrics derived from these measurements show how well both individual and group software quality assurance and control efforts are doing.

Metrics that measure the effectiveness of each of the actions implied by the metric, such as work product errors per function point, errors discovered per review hour, and errors discovered per testing hour, can be used. The defect removal efficiency (DRE) for each process framework activity can also be calculated using error data.

4.13.1 Measuring quality

Although there are numerous ways to gauge software quality8, the project team can utilize correctness, maintainability, integrity, and usability as valuable benchmarks. Gilb suggests definitions and measures for each

- Correctness: A program must function properly in order to be useful to its users. The degree to which the software fulfils its necessary purpose is known as correctness. Defects per KLOC, where a defect is defined as a validated lack of conformity to requirements, is the most popular metric for correctness. Defects are issues that a user of the program reports after the program has been made available for general use. These issues are taken into account when evaluating the overall quality of a software product. Defects are counted over a defined time period, usually one year, for quality evaluation purposes.

- Maintainability: Compared to other software engineering tasks, software maintenance and support need the most work. A program's maintainability refers to how easily it can be fixed when an error occurs, adjusted when its environment changes, or improved when the client requests a change in requirements. Since maintainability cannot be measured directly, indirect methods must be used. Mean-time-to-change (MTTC), which measures the time required to study a change request, develop an acceptable modification, implement the change, test it, and distribute the change to all users, is a straightforward time-oriented statistic. For equivalent types of modifications, maintainable programs often have a lower MTTC than unmaintainable programs.

- Integrity: In the era of online hackers and terrorists, software integrity has grown in importance. This characteristic evaluates a system's resistance to security threats, both unintentional and intentional. Programs, data, and documentation are the three elements of software that are vulnerable to attacks.

Threat and security are two additional variables that must be defined in order to quantify integrity. Threat is the likelihood that an attack of a particular type will take place within a certain period of time (which can be estimated or inferred from empirical evidence). Security is the likelihood that an assault of a particular type will be thwarted, which can be calculated or determined from empirical evidence. Thus, the definition of a system's integrity is

$$\text{Integrity} = \Sigma\, [1 - (\text{threat x } (1 - \text{security}))]$$

- Usability: Even if a program performs valuable functions, it is typically destined to failure if it is difficult to use. Usability is an attempt to measure usability.

4.13.2 Defect removal efficiency

Defect removal efficiency is a quality indicator that benefits both projects and processes (DRE). DRE essentially measures the effectiveness of quality assurance and control actions as they are applied throughout all activities governed by the process framework.

When taken into account for a project as a whole, DRE is described as follows:

$$DRE = [\ E / (E + D)]$$

where E is the number of mistakes discovered before to the software being delivered to the end user and D is the number of flaws discovered following delivery.

DRE should be set to a value of 1. In other words, the software has no flaws. Realistically, D will be bigger than 0, but as E rises for a particular value of D, the value of DRE can still get closer to 1. In reality, it is likely that the final value of D will decrease as E increases (errors are filtered out before they become defects). DRE urges a team working on software projects to use methods for locating as many errors as feasible before delivery if used as a metric that shows the effectiveness of quality control and assurance efforts.

DRE may also be used inside to a project to evaluate a team's capacity to identify mistakes before they are forwarded to the following framework or software engineering activity. A requirements model, for instance, is created by a requirements analysis and may be checked over to identify and fix flaws. The design phase is where any errors that were not discovered during the requirements model review may or may not be discovered. DRE is redefined as follows in this context:

$$\text{DRE}_i = \frac{E_i}{E_i + E_{i+1}}$$

where E_i is the total number of mistakes made in software engineering action I and E_{i+1} is the total number of mistakes made in software engineering action I + 1 that can be linked to mistakes made in software engineering action i. Achieving DRE_i that is close to 1 is a quality goal for a software team (or a single software developer). In other words, faults ought to be caught before being passed on to the subsequent activity or action.

SUMMARY

You can evaluate quality before the product is produced thanks to software metrics, which offer a quantifiable technique to evaluate the quality of internal product attributes. Metrics give you the knowledge you need to produce effective requirements and design models, reliable code, and exhaustive tests. A software measure needs to be straightforward, calculable, compelling, consistent, and objective in order to be helpful in real-world settings. It should be independent of the programming language you're using and give you useful feedback.

Function, data, and behavior: the model's three component are the main metrics for the requirements model. Design metrics take into account concerns with architecture, component-level design, and interface design. Metrics for architectural design take into account the model's structural elements. By creating proximate measures for cohesion, coupling, and complexity, component-level design metrics give an indicator of module quality.

At the source code level, Halstead gives a fascinating collection of metrics. Software science offers a range of metrics to evaluate program quality based on the number of operators and operands contained in the code. There aren't many product metrics that have been directly suggested for use in software testing and maintenance. The testing process can be guided by a variety of additional product criteria, which can also be used to evaluate a computer program's maintainability. The testability of an OO system has been evaluated using a wide range of OO metrics.

LIST OF REFERENCES

1] Software Engineering, A Practitioner's Approach, Roger S, Pressman (2014).

2] Software Engineering, Ian Sommerville, Pearson Education.

3] Fundamentals of Software Engineering, Fourth Edition, Rajib Mall, PHI.

4] Software Engineering: Principles and Practices, Hans Van Vliet, John Wiley & Sons.

5] A Concise Introduction to Software Engineering, Pankaj Jalote, Springer.

UNIT END EXERCISES

1] What are the aspects of product metrics?

2] Explain the terminologies: Measures, Metrics and Indicators.

3] What do you mean by function-based metrics?

4] Write a note on metrics for object-oriented design.

5] Explain operation-oriented metrics.

6] What do you mean by user interface design metrics?

7] Discuss on metrics for source code.

8] Explain the Halstead metrics applied to testing.

9] Write a note on metrics for maintenance.

10] What do you mean by cyclomatic complexity? Discuss with examples to find the cyclomatic complexity of the graph.

11] Explain size-oriented metrics.

12] Discuss on function-oriented metrics.

13] Write a note on metrics for software quality.

14] Explain the term: Measuring quality.

15] Discuss the concept of defect removal efficiency.

❄❄❄❄❄❄❄

5

SOFTWARE PROJECT MANAGEMENT

Unit Structure :

5.0 OBJECTIVES

- To understand the process involve in project planning
- To get acquaint with the scope and feasibility related to the software
- To get familiar with different estimation model and their workflow

5.1 INTRODUCTION

Project planning, a collection of related tasks, serves as the foundation for software project management. The software team must determine how much work will need to be done, what resources will be needed, and how much time it will take for completing the reckon before it can start. After completing these steps, the firmware management group must create a project schema outlining the firmware highlights, assigns accountability for each job, and details any inter-task dependencies that could significantly affect progress.

Steve McConnell offers a practical perspective on project planning in his outstanding manual for "software project survival":

Many technical personnel would prefer to do technical tasks than to prepare. Many technical managers lack the technical management training necessary to be confident that their planning will enhance the success of a project. No one wants to plan; thus, it frequently doesn't get done.

Efficient organization is required for handling issues upstream than downstream at high expense. However, failing to prepare is one of the most crucial mistakes a project can make. Rework, or correcting errors from earlier in the project, takes up an average of 80% of a project's time.

5.2 ESTIMATION IN PROJECT PLANNING PROCESS

Goal is to give the manager a complete schema so they can estimate resources, costs, and schedules in a fair manner. In order for project outcomes to be bounded, estimates should also make an effort to specify best scenarios and worst phase situations. The firmware group starts out on objectives formed as a result of these occupants, despite fact that there is a certain amount of inherent unpredictability. Therefore, as the project advances, the schema should be modified and refurbished.

Iterative planning begins with the creation of a starting schema in the launching stage. Fig 1 depicts a project planning process' framework. Plans will inevitably alter. You should routinely amend to subjective necessities, plan of action, and threaten measures as additional details about the module and the group come to light for the execution stage. Project plans alter as a result of shifting corporate objectives. Any initiatives that are affected by shifting corporate objectives may need to be rescheduled.

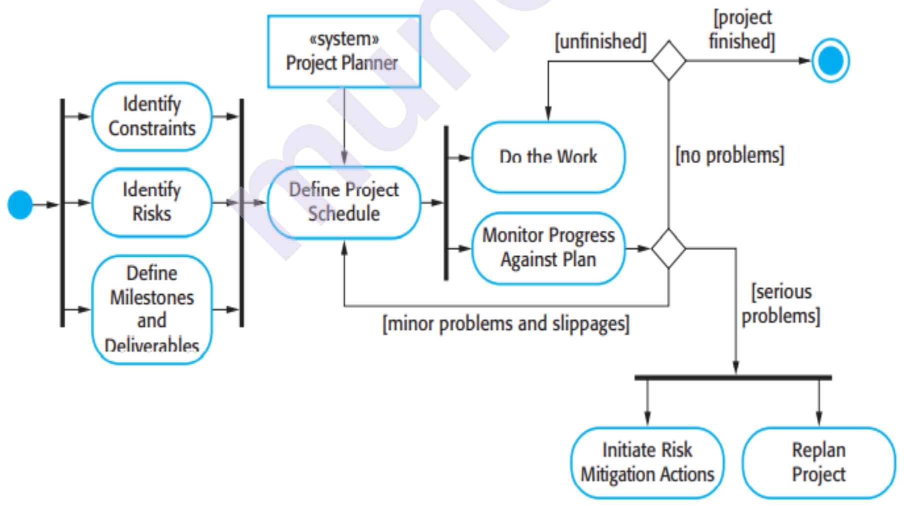


Figure 1: Project planning process

You should evaluate the project's limits before commencing a planning process. These limitations include the deadline for delivery, the number of employees on hand, the overall budget, the tools at hand, and others. One must specify the glimpses and highlights in conjunction with this. Milestones are dates on the timetable that can be used to gauge progress, such as when the system is turned over for deployment.

The procedure loops back on itself. You create an estimated project timetable, and the tasks outlined in it are started or given the go-ahead to continue. You should examine your work after certain time stamp & make notice of any deviations from the original timeline. Few of the changes are common and one must need to modify the actual framework because early estimations of project parameters are inherently approximate.

When drafting a project plan, it's crucial to be practical. During a project, some sort of issue almost always arises, and this might cause project delays. So, instead of being optimistic, your first assumptions and scheduling should be pessimistic. Your plan should include enough contingency restrictions through the cycle.

You must start risk mitigation steps if any substantial issues with the enlargement task expected causing a considerable detain in order to lower the risks of project failure exists. Along with these steps, the project needs to be replanned. Renegotiating the project's restrictions and deliverables with the client may be necessary for this. Additionally, a new timeline for when the job should be finished must be devised and approved by the client.

You should set up a formal project technical review if the measures are insufficient. The goals of this review are to discover a different strategy enabling the task to proceed & to determine whether the task, its estimate, the customer's aim, and the firmware goals are in alignment.

A review may result in the recommendation to halt a task. This could be the outcome of managerial or technological errors but frequently results from outside changes that have an impact on the project. Large software projects can take several years to create. The business's goals and priorities will unavoidably alter during that time. These adjustments can indicate that firmware is not necessary or that the original needs are insufficient.

5.3 SOFTWARE SCOPE AND FEASIBILITY

The term "software scope" refers to a system's performance, limitations, interfaces, and dependability as well as the features and functions that must be provided to users as well as the i/p & o/p information along with the "content" the users see as a result of using the system. One of two methods is used to define scope:

1. Following discussions with all stakeholders, a chronicle explanation of the firmware span is created.

2. End users create a collection of use cases.

Prior to the start of estimation, the functions outlined in the context assessed and, in few of the instances, improved in offering information. Due to functional orientation estimations, some level of decomposition is frequently helpful. Processing and reaction times are taken into account while evaluating performance. Constraints are boundaries imposed on the software by other systems already in place, available memory, or external hardware.

After the scope is determined (with the customer's approval), it’s appropriate in inquiring, "Can we design firmware to satisfy the described projection? Can it be completed? Frequently managers force software engineers to skip through these questions, which leads to them being bogged down in a project that is doomed from the start. When they write, Putnam and Myers address this problem.

No matter how ephemeral it might seem to outsiders, not everything that can be imagined is possible, not even in software. Contrarily, software viability has four dependable dimensions:

- Technology: Using current technology, can a project be completed? Is it up to date with technology? Can flaws be minimized to a level appropriate for the application?
- Financial viability: Is it possible? Can the cost of development be kept within the means of firmware company, its user, or the space captured?
- Time: Will the project's time to market outperform that of the competition?
- Resources: Has the company acquires necessities for gaining profits and capturing the market?

Although sometimes disregarded, these are important step in the estimating process.

5.4 RESOURCE ESTIMATION

Estimating the necessities required for completion of the software development project is the second planning step. Figure 2 shows the development environment, reusable software components, and people as 3 main criteria of SE (hardware and S/W tools). Four features are listed for each resource: a detail illustrating the requirement, a list of available quantities, the time required for resource estimation, & the number of instances it get used. You might think of the final two attributes as a time frame. The availability should be determined as soon as is practically possible.

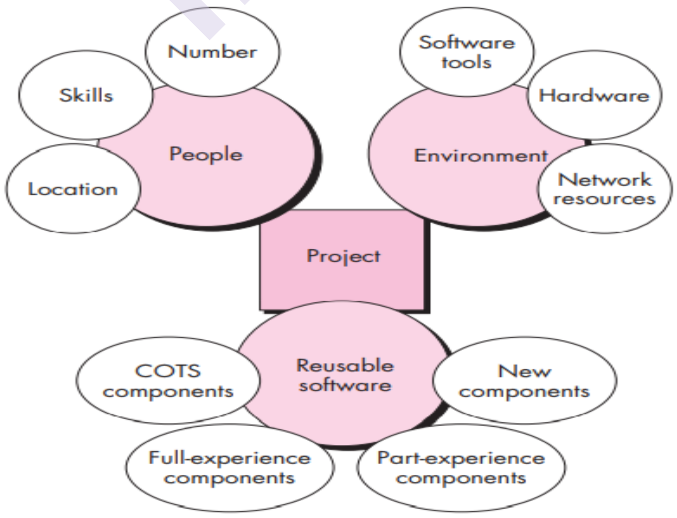


Figure 2: Project resources

5.4.1 Human resources

One must start through assessing the firmware extent and choosing required talents necessary for finishing the deployment. Twain organisational orientation and the specificity are mentioned. One person may do all software engineering activities consulting with experts as needed. The software crew may be geographically scattered over several distinct sites for larger projects. As a result, each human resource's location is given. Only once a development effort estimate (e.g., person-months) is developed can the required staff for a software project be estimated.

5.4.2 Reusable software resources

Reusability - the production and usage of software building blocks is emphasised by component-based software engineering (CBSE). Such building blocks, also known as components, need to be standardised for easy use, validated for easy integration, and catalogued for easy reference. As planning moves forward, Bennatan recommends taking into account the following four categories of software resources:

Components available from stores: existing software that is available from a previous project or from a third party. Components that are COTS (commercial off-the-shelf) are bought from arbitrator, completely vetted, and prepared for usage on the present project.

Full experience elements: Past projects' specifications, designs, codes, or test data comparable to firmware that is developed for present task. Protuberance of the ongoing firmware development group has extensive circumstance in the domain they represent. As a result, the chance of adjustments needed will be reduced.

Fragmentary experience modules: Partially developed requirements, frameworks, codes, or deployment information that are connected to the software to be developed for the present project but will need to be significantly modified. The current software team has only a little amount of expertise working with the application domain that these components represent. Therefore, there is a moderate amount of risk associated with adjustments needed for partial-experience components.

New components: The software team must create new software components expressly to meet the demands of the ongoing framework.

Recyclable firmware modules, ironically frequently overlooked across planning but addresses a top priority later in the firmware development procedure. Early software resource requirements definition is preferable. This allows for the technical assessment of the alternatives and prompt procurement.

5.4.3 Environmental resources

Software engineering environment (SEE), which enables firmware projects, combines both hardware and software. The firmware needed for creating the entities is a result of strong SE practise are supported by hardware. One should specify the formulation range necessary for H/W and S/W & ensure its availability.

The S/W group can need approaching to H/W pieces created by different group members. For instance, as part of the validation test phase, S/W used in a production environment could need particular robot according to its operation intended to perform. Planning must include the specification of each hardware component.

5.5 EMPIRICAL ESTIMATION MODELS – COCOMO II

To aid in estimating the effort, timing, and expenses of a software project, a number of models have been put forth. An empirical model called COCOMO II was created by compiling data from numerous software applications. These data were examined in order to identify the formulas that best suit the observations. These equations related the effort to construct the system to the system's size as well as to project, team, and product factors. A well-documented and open-source estimating model is COCOMO II.

Earlier COCOMO cost estimating models, which mostly relied on original code creation, served as the foundation for COCOMO II. The COCOMO II model takes into account more contemporary methods of software development, including component-based development, quick development using dynamic languages, and database programming. The spiral model of development is supported by COCOMO II, which also incorporates sub models that result in ever-more-detailed estimations.

As shown in the figure 3, the sub models that are a part of COCOMO II model are:

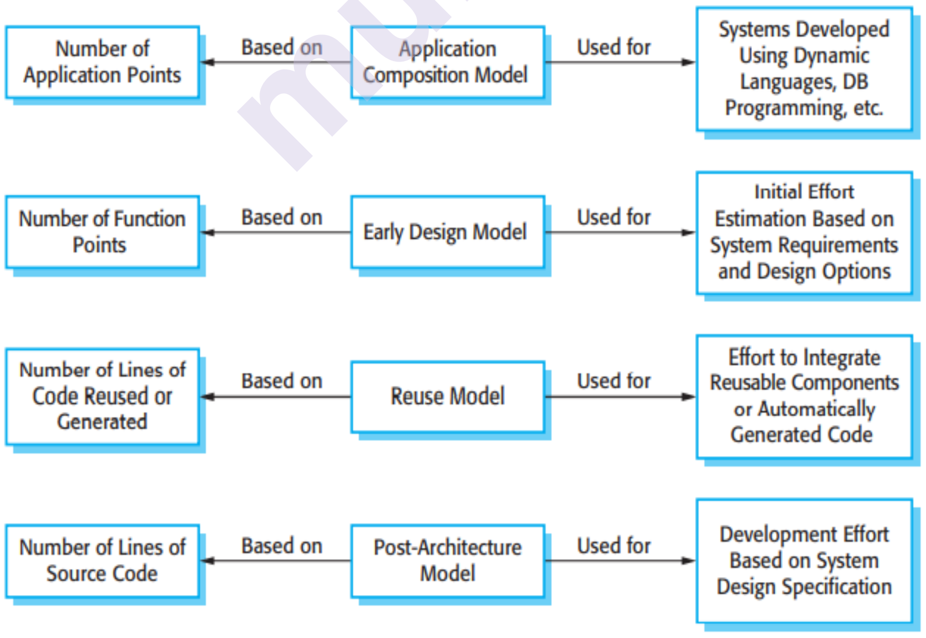

Number of Application Points
Based on
Application Composition Model
Used for
Systems Developed Using Dynamic Languages, DB Programming, etc.
Number of Function Points
Based on
Early Design Model
Used for
Initial Effort Estimation Based on System Requirements and Design Options
Number of Lines of Code Reused or Generated
Based on
Reuse Model
Used for
Effort to Integrate Reusable Components or Automatically Generated Code
Number of Lines of Source Code
Based on
Post-Architecture Model
Used for
Development Effort Based on System Design Specification

Figure 3: COCOMO estimation models

1] An application-composition model: This represents the effort needed to design systems made up of scripting, database programming, or reusable components. Application points are utilised to evaluate software size, and a straightforward size/productivity calculation is employed to calculate the work needed. A program's application points are a weighted average of the number of distinct screens that are shown, the number of reports generated, the number of modules in imperative programming languages (like Java), and the number of lines of scripting language or database programming code.

2] An early design model: After the requirements have been identified, this model is employed in the early stages of the system design. With a condensed set of seven multipliers, the estimate is based on the common estimating formula I covered in the introduction. The number of source code lines are translated from function points, on which the estimates are based, into function points. A language-independent method of measuring programme functionality is using function points. You can determine how many external inputs and outputs, user interactions, external interfaces, and files or database tables that the system uses by counting or estimating them.

3] A reuse model: It is used to determine how much work is involved in integrating reusable parts and/or automatically generated computer code. It frequently functions in tandem with the post-architecture model.

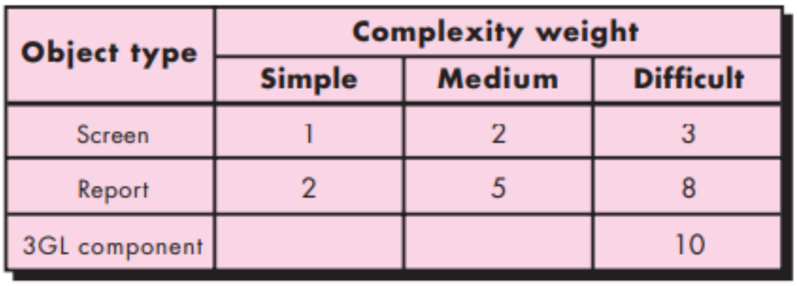
4] A post-architecture model: After the system architecture is created, it is possible to determine the software size with more accuracy. It includes a larger set of 17 multipliers that reflect project, product, and personnel aspects.

Of However, with huge systems, not every component needs to be estimated with the same level of precision because different system components may have been designed using various technologies. In such circumstances, you can combine the findings to get a composite estimate by using the appropriate sub model for each component of the system.

Following the determination of complexity, the number of screens, reports, and components is weighted in accordance with the table shown in Figure 4. The object point count is then calculated by averaging the overall object point count after multiplying the initial number of object instances by the weighting factor in the figure. The object point count is updated and the percent of reuse (%reuse) is estimated when component-based development or generic software reuse is to be used:

$$\text{NOP} = (\text{object points}) \times [(100 - \%\text{reuse})/100]$$

Where, NOP = new object points



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

Figure 4: Complexity weighting for object types

To derive an estimate of effort based on the computed NOP value, a "productivity rate" must be derived

$$\text{PROD} = \frac{\text{NOP}}{\text{person-month}}$$

for different levels of developer experience and development environment maturity.

Once the productivity rate has been determined, an estimate of project effort is computed using

$$\text{Estimated effort} = \frac{\text{NOP}}{\text{PROD}}$$

In more advanced COCOMO II models, a variety of scale factors, cost drivers, and adjustment procedures are required.

5.6 ESTIMATION FOR AGILE DEVELOPMENT

Due to the fact that an agile project's needs are established by a collection of user scenarios (such as the "stories" in Extreme Programming), it is possible to create an estimation method that is informal, moderately disciplined, and useful for project planning for each software increment. Agile projects estimate using a decomposition method that includes the following steps:

1] For estimation purposes, each user scenario - the project's equivalent of a miniature use case created by end users or other stakeholders at the outset is taken into account separately.

2] The collection of software engineering tasks that will be necessary to create the scenario are broken down.

3a] Each task's effort requirement is estimated separately. Note: An estimation may be supported by empirical modelling, historical facts, or "experience."

3b] Alternatives include estimating the "volume" of the scenario in LOC, FP, or another volume-oriented measure (e.g., use-case count).

4a] To build an estimate for the scenario, the estimates for each task are added up.

4b] Alternatively, using historical data, the volume estimate for the scenario is converted into effort.

5] The effort estimates for a given software increment is created by adding the work estimates for all of the scenarios that need to be implemented.

This estimation approach serves two functions because the project length needed to produce a software increment is fairly brief (usually three to six weeks):

(1) to ensure that the number of scenarios included in the increment is in accordance with the resources available, and

(2) to create a framework for dividing up work as the increment develops.

5.7 THE MAKE/BUY DECISION

It is frequently more economical to buy software than to produce it in many software application domains. Software engineering managers must decide whether to make or buy, which can be compounded further by a variety of acquisition options:

(1) Off-the-shelf software can be purchased (or licenced),

(2) Software components with "full-experience" or "partial-experience" may be purchased, modified, and combined to fulfil particular demands, or

(3) A third option is for an outside contractor to create software specifically to the buyer's requirements.

The criticality of the software to be purchased and the final cost determine the phases involved in software acquisition. In some circumstances (such as low-cost PC software), it is less expensive to make a one-time purchase and experiment than it is to carry out a thorough study of available software options. In the end, the choice to make or buy is dependent on the following factors:

(1) Will the software product's delivery date be earlier than the date for internally generated software?

(2) Will the price of customisation and acquisition be less than the price of in-house software development?

(3) Will external support (such a maintenance contract) be cheaper than internal support? Each of the acquisition possibilities is subject to these requirements.

5.7.1 Creating a decision tree

Statistical methods like decision tree analysis can be added to the just-described procedures. Figure 5 shows a decision tree for software-based system X as an illustration. The software engineering organisation can in this situation

1) build system X from the scratch,

2) To build the system, reuse existing partial-experience components,

3) Purchase a pre-existing software item and change it to match local requirements, or

4) assign the software development to a third-party contractor

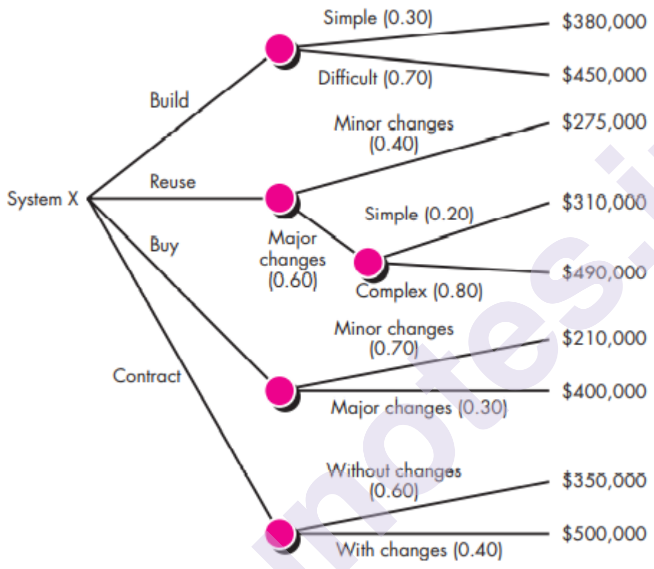


Figure 5: Decision-tree to support the make/buy decision

A 70% chance exists that the task of creating the system from scratch will be challenging. The project planner calculates that a challenging development endeavour will cost \$450,000 using the estimation approaches. The anticipated cost of a "basic" development project is \$380,000. Calculated along any decision tree branch, the predicted value for cost is

$$\text{Expected cost} = \Sigma\ (\text{path probability})_i \times (\text{estimated path cost})_i$$

where i is the decision tree path. For the build path,

$$\text{Expected cost}_{\text{build}} = 0.30\ (\$380\text{K}) + 0.70\ (\$450\text{K}) = \$429\text{K}$$

Following other paths of the decision tree, the projected costs for reuse, purchase, and contract, under a variety of circumstances, are also shown. The expected costs for these paths are

$$\text{Expected cost}_{\text{reuse}} = 0.40\ (\$275\text{K}) + 0.60\ [0.20\ (\$310\text{K}) + 0.80\ (\$490\text{K})] = \$382\text{K}$$
$$\text{Expected cost}_{\text{buy}} = 0.70\ (\$210\text{K}) + 0.30\ (\$400\text{K}) = \$267\text{K}$$
$$\text{Expected cost}_{\text{contract}} = 0.60\ (\$350\text{K}) + 0.40\ (\$500\text{K}) = \$410\text{K}$$

The "purchase" option has the lowest predicted cost based on the probability and projected costs shown in Figure. It's crucial to remember, though, that while making a purchase, a variety of factors more than just price must be taken into account. A few factors that could influence whether to build, reuse, buy, or contract include availability, experience of the developer, vendor, or contractor, compliance to specifications, local "politics," and the chance of change.

5.7.2 Outsourcing

Every business that creates computer software eventually wonders the same fundamental question: "Is there a way we can get the software and systems we need for a lesser price?" The answer to this question is not straightforward, and the heated debates that follow the subject always come down to one word: outsourcing.

Outsourcing is quite straightforward in theory. A third party is hired to perform software engineering tasks at a reduced cost and, ideally, a higher standard. A company's internal software development is really just contract management.

The choice to outsource might be tactical or strategic. Business managers analyse whether a sizable fraction of all software work may be outsourced at the strategic level. A project manager decides whether subcontracting the software work is the most effective way to complete all or a portion of a project at the tactical level. Regardless of the scope of the decision, outsourcing is frequently a financial one.

On the bright side, lowering the number of software employees and the infrastructure (such as computers) that supports them typically results in cost savings. On the down side, a business loses some control over the necessary software. A business risks the danger of entrusting a third party with the fate of its competitiveness because software is a technology that distinguishes its systems, services, and products.

Without a doubt, the trend toward outsourcing will persist. The only way to stop the trend is to acknowledge how fiercely competitive software work is at all levels. The only way to survive is to match the outsourcing providers' level of competition.

SUMMARY

Before a project starts, a software project planner must make three estimations: how long it will take, how much work it will require, and how many people it will involve. The planner must also forecast the risk involved as well as the resources (hardware and software) that will be needed.

Various graphical representations of the project plan are created as part of the scheduling process. The most popular schedule representations are bar charts, which display activity duration and staffing timelines.

A project milestone is an expected result of a task or series of tasks. A documented report of progress should be given to management at each milestone. A deliverable is a piece of work that is given to the project's client.

We also talked about how the COCOMO II costing model is an advanced computational cost model that incorporates project, product, hardware, and employee attributes when estimating costs.

At least two of the three aforementioned methods are often used to produce accurate project estimates. The planner is more likely to arrive at an accurate estimate by comparing and reconciling estimates created using several methodologies. Although software project estimation will never be a precise science, it can be made more accurate by using a combination of reliable historical data and methodical procedures.

LIST OF REFERENCES

1] Software Engineering, A Practitioner's Approach, Roger S, Pressman (2014).

2] Software Engineering, Ian Sommerville, Pearson Education.

3] Fundamentals of Software Engineering, Fourth Edition, Rajib Mall, PHI.

4] Software Engineering: Principles and Practices, Hans Van Vliet, John Wiley & Sons.

5] A Concise Introduction to Software Engineering, Pankaj Jalote, Springer.

UNIT END EXERCISES

1] Explain the estimation in project planning process.

2] Write a note on software scope and feasibility.

3] Discuss on resource estimation.

4] What are the factors included in human resources?

5] Explain in brief the concept of reusable software resources.

6] What are environmental resources?

7] Discuss COCOMO II model in detailed.

8] What are the fundamentals of estimation for agile development?

9] Explain the process of make/buy decision.

10] How will you create a decision tree. Explain suing concept of make/ buy decision.

11] Explain the outsourcing involved in make/buy decision.

❄❄❄❄❄❄❄

6

PROJECT SCHEDULING

Unit Structure :

6.0 OBJECTIVES

- To understand the workflow involved in project scheduling
- To get familiar with the principles involved in software engineering
- To get acquaint with the relationships and their interconnectivities with respect to the project scheduling
- To know the steps and the outline procedure associated with project scheduling

6.1 INTRODUCTION

When software projects run behind schedule, Fred Brooks was once questioned about it. One day at a time, was his profound though understated reply.

The actual scenario of a technological task is that thousands of minor jobs must be completed in order to achieve a greater goal, whether it is developing an operating system or building a hydroelectric facility. Some of these jobs are not commonplace and can be completed without worrying about how it will affect the project's deadline. On the "critical route," there are other tasks. The project's overall completion date is in peril if these "essential" tasks are delayed.

As a opportunity team leader, the goal is in declaring all the modules and its sub components, develop a strong connections describing the interrelationships, analyse the modules that are most important & combat the achievable. To achieve all the above-mentioned criteria's, one should maintain a detailed timetable so that the outcomes can be measured at every time stamp.

S/W project scheduling is the process of distributing estimated effort by assigning the effort to particular software engineering jobs over the course of the anticipated project duration. But it's crucial to remember that the schedule changes throughout time. A macroscopic timetable is created in the initial stages of project planning. This kind of schedule lists all significant activities that make up the process framework. Each item on macroscopic timetable is transformed into a detailed schedule as the project progresses. Here, precise software tasks and actions that must be completed in order to complete an activity are scheduled.

There are two very distinct ways to approach scheduling for software engineering projects. In the first, a final release date for a desktop-based model has been decided upon (& cannot be changed). The S/W company confines at allocating resources cross the allotted framework. Another perspective presupposes the broad sequential constraints that are negotiated, but SE organisation sets the end date. An end date is established after comprehensive examination of the programme and distribution of effort is made to make the greatest use of available resources. Sadly, the first circumstance arises multiple times than the later scenario.

6.2 BASIC PRINCIPLES

Software project scheduling is governed by a few fundamental rules, just like all other aspects of software engineering:

- Project compartmentalization: The project needs to be broken down into a number of doable tasks and activities. The process and the product are both improved to achieve compartmentalization.
- Interdependence: It is necessary to assess the interdependence of each segregated task or activity. While certain jobs must be completed in order, others can be completed concurrently. Some tasks can't start until someone else's finished result is accessible. Other things can happen on their own.
- Time assignment: Every module that needs to be managed must be given a certain amount of grind measure in terms of time. Additionally, every module needs to be given a beginning and an end period that rely on the alliance & if the task be done on time.
- Validating attempt: The software team for each project consists of a specific number of individuals. You must make sure that allotted nos. of person are organized and managed at particular moment as assigned. Take a forecast with 3 S/W engineers as an example. Seven concurrent tasks must be completed on any given day. It takes 0.50 person-days to complete each activity. There are more persons assigned to the task than there are available workers.

- Assigned liability: Each subsystem organized ought to be given a particular group leader.
- Assigned objectives: Each work that is organized must be clearly stated objective. The end result of software projects often consists of a product or its counterpart. Deliverables frequently integrate sub modules.
- Declared highlights: Each module, or set of modules, must be connected to specific glimpse or highlights. When multiple by-products have undergone a qualitative evaluation & approval, a milestone is reached.

6.3 RELATIONSHIP BETWEEN PEOPLE AND EFFORT

One person can evaluate requirements, carry out design, create code, and run tests in small S/W expansion system. A project requires more participation as it grows in size. (We hardly ever have the splendour of doing a 10-people attempt with a single individual doing for 10 yrs)

Many managers in charge of software development projects still hold fast to the popular misconception that "if we are way back than our timeline, we recruit more employees and try to come up to the level of meeting the specifications". Unintentionally, employing more personnel at the end of a project frequently disrupts it and pushes back deadlines. The newly added individuals must learn the system, and those who were performing the work are also the ones who are instructing them. Since no work is done while lecturing, the project is further behind schedule.

Many employees escalate the amount of divulgence pathways & the difficulty of conveyance in a system, which adds to the effort & schedule set required to master the modules. Although effective communication is crucial for the creation of good software, every new communication line involves more work, which adds time.

Project timetables are flexible, as shown by empirical data and theoretical study over time. In other words, a planned project completion date can be somewhat shortened (by adding more resources). It makes feasible in postponing a deadline (by making less use of materials).

A software project's association between endeavour put out and time required to achieve a specified outcome is depicted by the Putnam-Norden-Rayleigh (PNR) Curve. Figure 1 depicts a variant of the curve that plots project effort against delivery time. It shows minimal quantity t_o indicating least delivery expenditure. The curve climbs nonlinearly moving towards left side of t_o (i.e., striving hard to speed up dispatch process).

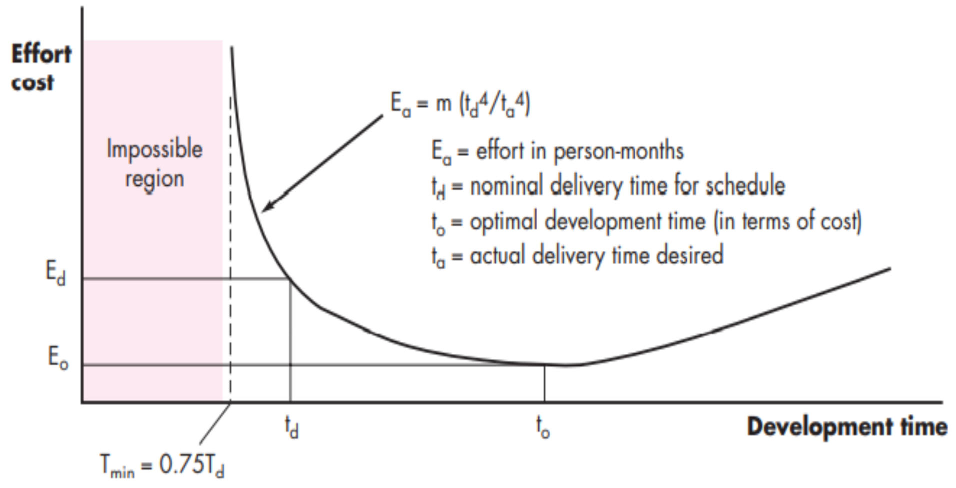


Figure 1: Association of effort & dispatch time

Assume that a manager involved in the task has calculated the amount of endeavour Ed necessary in accomplishing a usual dispatch time td which is ideal with respect to the schedule and resource availability. Although delivery can be sped up, the graph increases quite abruptly towards left of t_d. In reality, the PNR curve suggests that beyond $0.75t_d$, the amount of quantifiable outcome measurement is significantly reduced. The project enters "the impracticable domain" and the chance of collapse increases if we attempt any further compression. The PNR curve also shows that to = $2t_d$ is the least expensive delivery choice. The implication is that postponing project completion can drastically lower expenses. Of course, this needs to be compared to the lost revenue caused by the obstruction.

The S/W formulation, obtained from through PNR curve, illustrates extremely unpredictable and irregular association between the amount of time required to accomplish a task chronologically & the amount of labour put into it. The following equation relates effort and development time to the supplied lines of code (source statements), L:

$$L = P \times E^{1/3}t^{4/3}$$

where P is a productivity measure that reflects a number of elements that contribute to high-quality software engineering work (average values for P vary between 2000 and 12,000), E is development effort in person-months, and t is the project duration in calendar months.

The above S/W formulation can be rearranged to yield an objective function and a mathematical formulation for enhancement endeavour E:

$$E = \frac{L^3}{P^3t^4}$$

In above scenario t implies development period in terms of years and E is the endeavour put forth during a course of the software development and maintenance life cycle. By including a encumber labour cost component ($/person-year), the formulation for enhancement endeavour and cost will be connected.

This produces some intriguing outcomes. Assume a challenging real-time S/W task that would require 12 person-years and 33,000 LOC. The project can be finished in roughly 1.3 years if the project team consists of eight persons. The grater uncertain and unpredictable character of the systeml stated in the aforementioned formulation, however, results in:

$$E = \frac{L^3}{P^3 t^4} \sim 3.8 \text{ person-years}$$

This suggests that we can lower the number of participants from eight to four by delaying the finish date by six months! The veracity of these results is debatable, but it is evident that using fewer people for a little bit longer to achieve the same goal can be advantageous.

6.4 EFFORT DISTRIBUTION

The work units (such as person-months) necessary to finish software development are estimated using each of the software project estimation approaches. The 40-20-40 rule is a recommended way to distribute effort throughout the software development lifecycle. The front-end analysis and design portion of the project receives 40% of the total effort. Back-end testing uses a comparable percentage. You are correct to assume that deemphasis on code development (20% of work) is present.

Only use this effort distribution as a general reference. The distribution of work is determined by the specifics of each project. Unless the plan commits an organisation to significant expenditures with high risk, work put into project planning rarely accounts for more than 2 to 3 percent of effort. 10 to 25% of the project effort may be devoted to customer interaction and requirements analysis. The amount of effort put into analysis or prototyping should grow in direct proportion to the size and complexity of the project. Typically, software design requires 20 to 25 percent of the labour. You must also take into account the time needed for design review and future iterations.

The work put into software design should make it reasonably easy for code to follow. It is possible to attain a range of 15 to 20 percent of total effort. Debugging after testing might take up to 40% of the time spent developing software. The quantity of testing necessary is frequently determined by the software's criticality. Even higher percentages are normal if software is human graded, meaning that failure of the software could lead to fatalities.

6.5 TIME-LINE CHARTS

The work breakdown structure is the first collection of tasks you use to start a software project schedule. The work breakdown is entered as a task network or task outline if automated technologies are employed. Next, each task's effort, duration, and start date are entered. Tasks may also be delegated to particular people.

A time-line chart, often known as a Gantt chart, is produced as a result of this input. For the entire project, a timeline chart can be created. As an alternative, distinct flowcharts can be created for each project function or for each person involved.

The structure of a time-line chart is shown in Figure 2. It shows a section of a software project schedule that places emphasis on the work of concept scoping for a word-processing (WP) software application. The left-hand column contains a list of all project tasks (for concept scoping). The horizontal bars show how long each activity took. Task concurrency is implied when multiple bars appear on the calendar at the same time. Milestones are marked with diamonds.

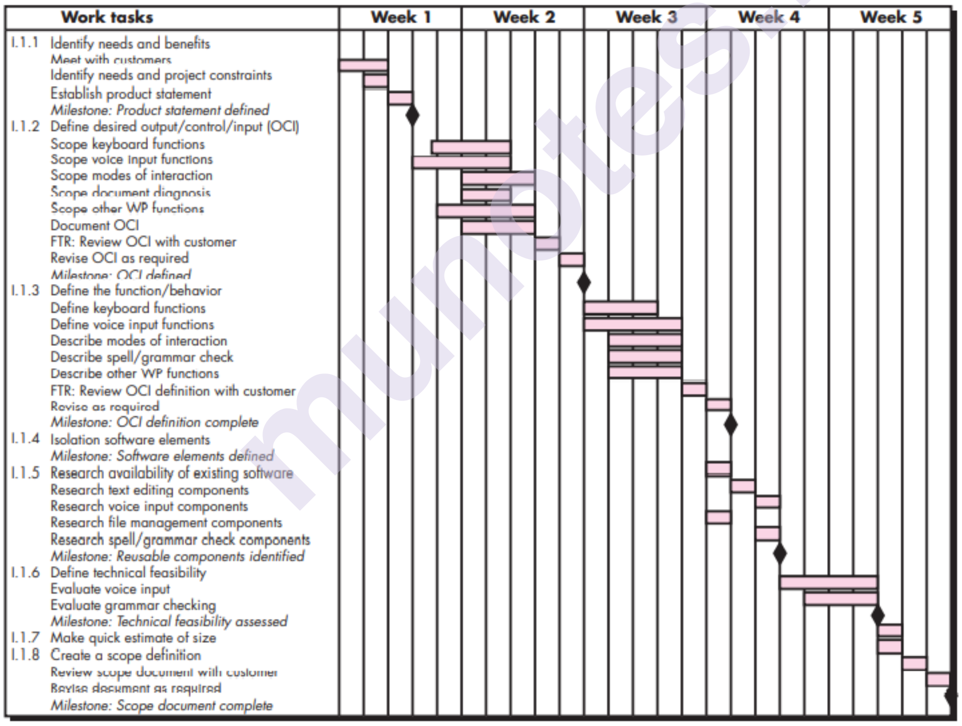
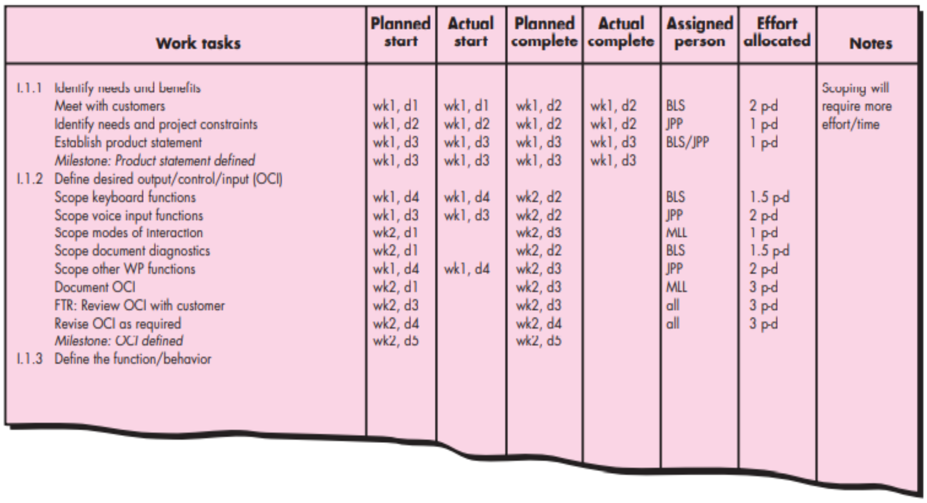


Figure 2: An example time-line chart

The majority of software project scheduling tools create project tables, a tabular listing of all project tasks, their planned and actual start and end dates, and various related information, once the data required for the creation of a time-line chart has been provided (Figure 3). You may monitor development by using project tables in conjunction with the timeline graphic.



Work tasks	Planned start	Actual start	Planned complete	Actual complete	Assigned person	Effort allocated	Notes
I.1.1 Identify needs and benefits							Scoping will require more effort/time
Meet with customers	wk1, d1	wk1, d1	wk1, d2	wk1, d2	BLS	2 p-d	
Identify needs and project constraints	wk1, d2	wk1, d2	wk1, d2	wk1, d2	JPP	1 p-d	
Establish product statement	wk1, d3	wk1, d3	wk1, d3	wk1, d3	BLS/JPP	1 p-d	
Milestone: Product statement defined	wk1, d3	wk1, d3	wk1, d3	wk1, d3			
I.1.2 Define desired output/control/input (OCI)							
Scope keyboard functions	wk1, d4	wk1, d4	wk2, d2		BLS	1.5 p-d	
Scope voice input functions	wk1, d3	wk1, d3	wk2, d2		JPP	2 p-d	
Scope modes of interaction	wk2, d1		wk2, d3		MLL	1 p-d	
Scope document diagnostics	wk2, d1		wk2, d2		BLS	1.5 p-d	
Scope other WP functions	wk1, d4	wk1, d4	wk2, d3		JPP	2 p-d	
Document OCI	wk2, d1		wk2, d3		MLL	3 p-d	
FTR: Review OCI with customer	wk2, d3		wk2, d3		all	3 p-d	
Revise OCI as required	wk2, d4		wk2, d4		all	3 p-d	
Milestone: OCI defined	wk2, d5		wk2, d5				
I.1.3 Define the function/behavior							

Figure 3: An example project table

SUMMARY

The conclusion of a planning activity, which is a crucial part of software project management, is scheduling. Scheduling provides the project manager with a roadmap when used in conjunction with estimating techniques and risk assessments.

Decomposing the process is the first step in scheduling. A suitable task set is modified based on the project's characteristics and the work that has to be done. Each engineering task, together with its dependence on other activities and its anticipated length, is represented by a task network. The critical route, a time-line chart, and other project data are computed using the task network. You may monitor and manage each phase of the software development process using the timetable as a reference.

LIST OF REFERENCES

1] Software Engineering, A Practitioner’s Approach, Roger S, Pressman (2014).

2] Software Engineering, Ian Sommerville, Pearson Education.

3] Fundamentals of Software Engineering, Fourth Edition, Rajib Mall, PHI.

4] Software Engineering: Principles and Practices, Hans Van Vliet, John Wiley & Sons.

5] A Concise Introduction to Software Engineering, Pankaj Jalote, Springer.

UNIT END EXERCISES

1] Explain the term project scheduling.

2] What are the basic principles involved in software project scheduling?

3] Explain the terms: Project compartmentalization, Interdependence and time allocation.

4] Discuss the term validating effort associated with project scheduling.

5] What do you mean by Defined responsibilities, defined objectives and defined milestones.

6] Write a note on relationship between people and effort.

7] With the help of suitable diagram explain the relationship between effort and delivery time.

8] What do you mean by effort distribution?

9] What are time-line charts? Illustrate with suitable figure

❄❄❄❄❄❄❄

7

RISK MANAGEMENT

Unit Structure :

7.0 OBJECTIVE:

After going through this unit, you will be able to:

- Understand what Software Risk is?
- Define risk projection and risk refinement.
- Know about RMMM

7.1 INTRODUCTION:

Risk is a problem that could origin some loss or hover the progress of the project, but which has not happened yet. These possible issues might harm cost, schedule or technical attainment of the project and the quality of our software device, or project team confidence. Risk Management is the system of recognising addressing and abolishing these problems before they can damage the project. We need to distinguish risks, as potential issues, from the current problems of the project.

7.2 SOFTWARE RISKS

A software project can be alarmed with a large variety of risks. In order to be proficient to systematically identify the substantial risks which might affect a software project, it is necessary to classify risks into diverse classes. The project manager can then check which risks from each class are applicable to the project.

Software Risk Management is the process of identifying, evaluating, and mitigating potential risks that may affect the success of a software development project. The goal of software risk management is to reduce the negative power of risks and to ensure that the project is delivered on time, with the desired quality within budget and functionality.

There are three main classifications of risks which can affect a software project:

1. Project risks
2. Technical risks
3. Business risks

7.2.1. Project risks: Project risks concern differ forms of resource, schedule, budgetary, personnel, and customer-related problems. A vibrant project risk is plan slippage. Since the software is intangible, it is very hard to monitor and resistor a software project. It is always very tough to control something which dismiss to be identified. For any engineering program, such as the manufacturing of cars, the plan executive can identify the product taking shape.

7.2.2. Technical risks: Technical risks apprehension potential method, maintenance issue, testing, implementation, and interfacing. It also consists of an uncertain specification, inadequate specification, altering specification, technical ambiguity, and technical obsolescence. Most technical risks look like due to the development team's inadequate knowledge about the project.

7.2.3. Business risks: This type of risks (losing budgetary or personnel commitments, etc.) cover risks of building an excellent product that no one need.

7.2.4 Additional Risk categories:

a. **Schedule Risks:** These risks are associated to the timeline of the project. This type covers potential for postponements or missed deadlines.

b. **Resource Risks:** These risks are related to the obtainability and sharing of resources, such as personnel, funding, or equipment.

c. **Quality Risks:** These risks are associated to the quality of the software being developed, including the probable for bugs, security weaknesses, or user experience issues.

d. **Regulatory and Legal Risks:** These risks are related to legal and regulatory agreement issues, i.e. data privacy, intellectual property, or export controls.

e. **Identified risks:** This type of risks can be exposed after careful valuation of the project program, the business and technical environment in which the plan is being developed, and more reliable data sources (for example impractical delivery date)

f. **Expected risks:** Those risks that are assumed from previous project experience.

g. **Unpredictable risks:** These type of risks that can, and do occur, but are tremendously tough to identify in advance.

7.2.5 Methods for Identifying Risks:

There are some methods to plan for risk management:

- **Transfer the risk:** This method comprises the risky element developed by a third party, i.e. buying insurance cover, etc.
- **Avoid the risk:** This may take numerous ways such as discussing with the client to change the requirements to decrease the scope of the work. To give incentives to the resources to avoid the risk of human resources throughput, etc.
- **Risk decline:** This means scheduling method to include the loss due to risk. For example, if there is a risk that some key personnel might leave, new recruitment can be planned.

It is significant for software development teams to recognize and evaluate these risks, and to put justification strategies in place to minimize their effect on the project. This can include contingency planning, regular risk assessments, and risk management processes.

7.3 RISK IDENTIFICATION:

The first step in software risk management is to find probable risks that may influence the project. This may include technical risks, schedule risks, resource risks, quality risks, business risks, and legal and regulatory risks. Actual risk management begins with detecting and assessing risks, including vulnerabilities and potential threats, and arranging them based on their potential impact and prospect.

Previously, there were no easy procedures available that will surely detect all risks. But currently, there are some supplementary approaches available for classifying risks. Some of approaches for risk identification are as follows:

1. **Checklist Analysis** – Checklist Analysis is type of method generally used to detect or find risks and manage it effectively. The specification is basically developed by listing items, steps, or even tasks and is then further examined against criteria to just classify and determine if process is completed correctly or not. It is list of risk that is just found to happen regularly in progress of software project. Below is the list of software development risk by Barry Boehm-modified version.

Risk	**Risk Reduction Technique**
Personnel Shortages	Various methods include training and career development, job-matching, teambuilding, etc.
Unrealistic time and cost estimates	Various techniques include incremental development, standardization of methods, recording, and analysis of the past project, etc.
Development of wrong software functions	Various techniques include formal specification methods, user surveys, etc.
Development of the wrong user interface	Various techniques include user involvement, prototyping, etc.

2. **Brainstorming –** This procedure provides and gives free and exposed methodology that usually increases each and every one on project team to add. It also results in better sense of ownership of project risk, and team usually committed to dealing risk for given time period of project. It is creative and exclusive technique to gather risks freely by team members. The team members identify and govern risks in 'no wrong answer' atmosphere. This technique also delivers chance for team members to always improve on each other's ideas. This technique is also used to define best possible solution to difficulties and issue that rises and develop.

3. **Casual Mapping –** It is method that shapes or develops on replication and review of failure factors in reason and result of the diagrams. It is very useful for assisting learning with an organization or system simply as method of project-post assessment. It is also crucial tool for risk assessment.

4. **SWOT Analysis** – Strengths-Weaknesses-Opportunities-Threat (SWOT) is very important and helpful technique for identifying risks inside greater organization context. It is generally used as scheduling tool for analysing business, its resources, and also its atmosphere simply by looking at inside strengths and weaknesses, opportunities and threats in outer environment. It is technique often used in preparation of strategy. The suitable time and effort should be spent on thinking completely about faults and threats of organization for SWOT analysis to more effective and effective in risk identification.

5. **Flowchart Method –** This method permits for go-ahead process to be diagrammatically denoted in paper. This method is generally used to represent actions of process graphically and serially to simply identify the risk.

7.4 RISK PROJECTION AND RISK REFINEMENT

There are two essential steps in the course of software risk management - Risk Projection and Risk Refinement.

Risk Projection contains historical data and expert conclusion to estimate the prospect and impact of potential risks that may impact the software development project. This helps to arrange the risks and to assign resources and effort to address the most acute risks.

Risk Refinement involves apprising and refining the risk projections based on new information, changing situations, and the implementation of risk mitigation strategies. This helps to ensure that the risk management plan remains significant and effective during the software development lifecycle.

Risk Refinement may include reviewing the possibility and impact of risks, updating risk mitigation approaches, and re-evaluating the urgency of the risks. It may also involve observing the implementation of the risk management plan and gathering feedback from stakeholders to detect areas for improvement.

Risk Projection and Risk Refinement are serious to the success of software risk management because they support to ensure that the risk management plan remains related and effective throughout the software development lifecycle. By frequently updating and refining the risk managing plan, organizations can better formulate for potential risks and reduce the negative impact of risks on the project.

7.5 RMMM PLAN

RMMM (Risk Management, Monitoring, and Mitigation) Plan is a all-inclusive plan that summaries the method for assessing, identifying, and mitigating risks in software development projects. The RMMM plan helps as a roadmap for managing risks during the software development lifecycle and delivers a structured approach for certifying that risks are succeeded efficiently.

The RMMM plan usually includes the below components:

7.5.1 Risk Identification: This component outlines the procedure for identifying possible risks that may affect the software development project. This may include technical risks, schedule risks, resource risks, quality risks, business risks, and legal and regulatory risks.

7.5.2 Risk Assessment: This component shapes the process for evaluating and analysing the identified risks to control their impact and likelihood. This information is used to arrange the risks and to distribute resources and effort to report the most critical risks.

7.5.3 Risk Mitigation: This component summaries the strategies and arrangements to be taken to moderate the risks, such as decreasing the likelihood of the risk happening or reducing the influence if it does occur.

This may include developing possibility plans, allocating further resources, or altering the project method or schedule.

7.5.4 Risk Monitoring: This element plans the process for constantly monitoring and revising the risks to ensure that they are being managed excellently and to identify new risks as they rise. This may include regular risk assessments, stakeholder communication, and risk management reports.

7.5.5 Risk Evaluation: It supports the process for evaluating the RMMM plan after the project is completed to regulate its effectiveness and to find opportunities for improvement in upcoming projects.

The RMMM plan should be reviewed and restructured regularly during the software development lifecycle to certify that it remains appropriate and effective.

SUMMARY

Software development is an advanced activity that works a wide range of technological developments. Every software development project comprises elements of ambiguity due to these and other factors. The amount of risk connected with each project activity governs the success of a software development project. It is not enough to just be aware of the threats. To achieve success, project management must assess, prioritize, identify, and manage all foremost risks.

- Pressman, R. S. (2010). Software engineering: a practitioner's approach (7th ed.). McGraw-Hill.
- ISO/IEC 12207:2017 - Information technology — Software life cycle processes.
- IEEE Standard for Software Project Management Plans (IEEE 1058-1998).
- Boehm, B. W. (1981). Software Engineering Economics. Prentice-Hall.
- McConnell, S. (1996). Rapid Development: Taming Wild Software Schedules. Microsoft Press
- De Marco, T. (2002). Slack: Getting Past Burnout, Busywork, and the Myth of Total Efficiency. Broadway Business.
- Standish Group. (1994). Chaos report.
- Fink, A. L. (2002). Conducting literature reviews: From the Internet to paper. Sage.
- Clark, B., & Gorsky, P. (2010). A practitioner's guide to software risk management. John Wiley & Sons.
- Hazards, Risks and Disasters in Society. (2015). Butterworth-Heinemann.
- https://www.geeksforgeeks.org/methods-for-identifying-risks/

MODEL QUESTIONS:

- What is Software Risk? Explain different categories of Risks.
- How to identify Risk? Explain different techniques of risk Identification.
- Explain Risk Projection and Risk Refinement
- Describe RMMM Plan in detail.

❄❄❄❄❄❄❄

8

SOFTWARE QUALITY ASSURANCE

Unit Structure :

8.0 OBJECTIVES

After going through this unit, you will be able to:

- Understand what Software Quality Assurance is?
- Understand about task and Matrix of SQA.
- Know about ISO 9000 Quality standards and CMM.
- Apprehend Six Sigma.

8.1 INTRODUCTION

Quality states to any measureable characteristics such as accuracy, reliability, efficiency, maintainability, portability, testability, usability, integrity, reusability, and interoperability.

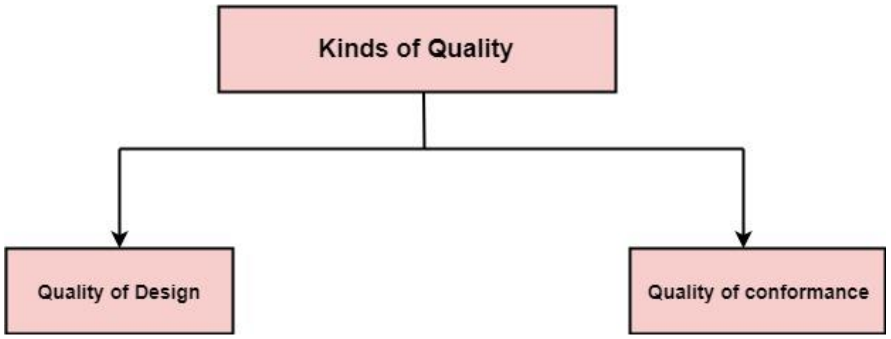
Software quality assurance is a strategic and systematic plan of all actions required to provide suitable confidence that an item or product conforms to create technical requirements. A set of activities considered to calculate the method by which the products are developed.

8.2 ELEMENTS OF SQA

Software quality assurance focus on the management of software quality using following elements.

- **Standards:** The ISO, IEEE and other standards groups have produced a broad range of software engineering standards and associated documents. Standards may be approved freely by a software engineering. The job of SQA is to certify that standards that have been approved are followed and that all effort products follow to them.
- **Reviews and audits:** Technical evaluations are a quality control activity executed by software engineers for their intent is to expose errors. Audits are a type of evaluation performed by SQA staffs with the intent of certifying that quality strategies are being followed for software engineering work.
- **Testing:** Software testing is a quality regulator function that has one primary goal "to find errors". The work of SQA is to certify that testing is conducted correctly and efficiently.
- **Error/defect collection and analysis:** SQA collects and analyses error and defect data to well understand how errors are familiarised and what software engineering activities are best suited to abolishing them.
- **Change management:** Change is one of the most unruly features of any software project. If it is not properly succeeded, change can lead to misunderstanding, and confusion practically leads to poor quality.
- **Education:** Every software organization wants to increase its software engineering practices. A vital contributor to upgrading is education of software engineers, their managers, and stakeholders.
- **Security management:** With the increase in cyber-crime and new government guidelines regarding privacy, every software group should institute policies that shelter data at all levels, establish firewall security for Web Apps, and ensure that software has not been damaged with internally.
- **Safety**: Because software is almost always a crucial component of human graded system. SQA may be responsible for calculating the impact of software failure and for originating those steps required to reduce risk.
- **Risk management:** SQA organization confirms that risk management actions are properly directed and that risk–related exigency plans have been established.

8.2.1 There are two kinds of Quality:



- **Quality of Design:** It refers to the characteristics that inventers specify for an item. The status of materials, acceptances, and performance provisions that all contribute to the quality of design.

- **Quality of conformance:** This is the degree to which the design specifications are followed during work. Greater the degree of conformance, the higher is the level of quality of conformance.

- **Software Quality:** Software Quality is distinct as the conformance to clearly state functional and performance supplies, clearly documented development standards, and natural characteristics that are projected of all professionally developed software.

- **Quality Control:** Quality Control comprises a series of inspections, reviews, and tests used during the software process to certify each work product meets the requirements place upon it. Quality control consist of a feedback loop to the process that formed the work product.

- **Quality Assurance:** Quality Assurance is the anticipatory set of activities that provide greater assurance that the project will be completed successfully.

8.2.2 Importance of Quality

As we expect the quality to be a concern of all manufacturers of goods and services. However, the distinct characteristics of software and in particular its intangibility and complexity, make superior demands.

- **Growing criticality of software:** The final customer or user is naturally worried about the general quality of software, especially its reliability. This is aggregate in the case as organizations become more dependent on their computer systems and software is used more and more in safety-critical areas.

- **The intangibility of software:** This makes it stimulating to know that a particular task in a project has been completed adequately. The results of these tasks can be made concrete by demanding that the developers produce 'deliverables' that can be inspected for quality.

8.3 SQA TASKS

Software Quality Assurance (SQA) comprises a number of tasks that are performed to confirm that software products meet the stated quality standards and requirements. Some of the crucial SQA tasks include:

- **Quality Planning:** This involves the development of a quality plan that summaries the activities, processes, and procedures that will be used to confirm software quality.

- **Requirements Analysis:** This includes the review and evaluation of the software requirements to ensure that they are complete, accurate, and steady.
- **Test Planning:** This involves the improvement of a test plan that outlines the testing activities, test cases, and test procedures that will be used to verify that the software meets the identified requirements.
- **Test Case Design:** This consist of the creation of test cases that are used to validate that the software works as planned.
- **Test Execution:** This involves the effecting of test cases to identify flaws in the software.
- **Test Reporting:** This implicates the documentation of test results and the identification of defects that need to be determined.
- **Defect Resolution:** This involves the purpose of defects identified during testing and the execution of corrective actions.
- **Configuration Management:** This involves the identification and control of the software artifacts and structures to ensure that the accurate versions are being used.
- **Process Evaluation:** This involves the estimate of the software development processes to ensure that they are actual and efficient.
- **Process Improvement:** This involves the implementation of continuous improvement events to enhance software excellence over time.
- **Audits:** This involves the free review of software processes to confirm that they adapt to the specified quality standards.
- **Metrics Collection and Analysis:** Involves the collection and exploration of software quality data to recognize areas for improvement.

8.4 - GOALS AND MATRICS

Following table shows Software quality goals, attributes, and metrics:

Goal	**Attribute**	**Metric**
Requirement quality	Ambigully	Number of ambiguous modifiers (e.., many, large, human–friendly)
	Totality	Number of TBA, TBD
	Understandability	Number of sections/subsections
	Volatility	Number of changes per requirement Time (by activity) when change is requested
	Traceability	Number of requirements not traceable to design/code

	Model clarity	Number of UML models Number of descriptive pages per model Number of UML errors
Design quality	Architectural integrity Component completeness Interface complexity Patterns	Existence of architectural model Number of components that trace to architectural model Complexity of procedural design Average number of pick to get to a typical function or content Layout appropriateness Number of patterns used
Code quality	Complexity Maintainability Understandability Reusability Documentation	Cyclomatic complexity Design factors (Chapter 8) Percent internal comments Variable naming conventions Percent reused components Readability index
QC effectiveness	Resource allocation Completion rate Review effectiveness Testing effectiveness	Staff hour percentage per activity Actual vs. budgeted completion time See review metrics Number of errors found and criticality Effort required to correct an error Origin of error

8.5 - FORMAL APPROACHES TO SQA:

There are several formal approaches to Software Quality Assurance (SQA) that organizations can use to ensure the delivery of high-quality software products. Some of the most commonly used formal approaches include:

- ISO/IEC 15504 (SPICE)
- CMMI (Capability Maturity Model Integration)
- ITIL (Information Technology Infrastructure Library)
- Six Sigma
- Agile Methods

Each of these formal approaches to SQA has its own strong point and faults, and organizations can choose the method that best fits their necessities based on the size and complexity of their software projects and their overall organizational culture and goals. By using a formal method to SQA, organizations can certify that their software development procedures are well-defined, effective, and efficient, and that they distribute high-quality software products to their customers.

8.6 - SIX SIGMA

Six Sigma is the procedure of improving the quality of the production by identifying and eliminating the cause of faults and reduce variability. The maturity of a manufacturing process can be defined by a sigma rating indicating its percentage of defect-free products it creates.

8.6.1 Characteristics of Six Sigma

The Characteristics of Six Sigma are as follows:

1. Statistical Quality Control
2. Methodical Approach
3. Fact and Data-Based Approach
4. Project and Objective-Based Focus
5. Customer Focus
6. Teamwork Method to Quality Management

8.6.2 Six Sigma Methodologies

Six Sigma projects carries two project methodologies:

1. DMAIC
2. DMADV

1. DMAIC

It states a data-driven quality strategy for enlightening processes. This methodology is used to enhance an existing business process.

The DMAIC project methodology has five phases:

1. **Define:** It covers the process plotting and flow-charting, project approval development, problem-solving tools.

2. **Measure:** It includes the principles of measurement, continuous and discrete data, and scales of measurement, an outline of the principle of variations and repeatability and reproducibility (RR) studies for continuous and discrete data.

3. **Analyze:** It covers creating a process baseline, how to determine process improvement goals, knowledge discovery, including descriptive and exploratory data analysis and data mining tools, the basic principle of Statistical Process Control (SPC), specialized control charts, process capability analysis,

correlation and regression analysis, analysis of categorical data, and non-parametric statistical methods.

4. **Improve:** It covers project management, risk assessment, process simulation, and design of experiments (DOE), robust design concepts, and process optimization.

5. **Control:** It covers process control planning, using SPC for operational control and PRE-Control.

2. DMADV

I t specifies a data-driven quality approach for designing products and processes. This method is used to generate new product designs or process designs in such a way that it results in a more expectable, mature, and discover free performance.

The DMADV project methodology has five phases:

1. **Define:** The problem or project goal that needs to be addressed.
2. **Measure:** It measures and defines the customer's needs and provisions.
3. **Analyze:** It analyses the method to meet customer needs.
4. **Design:** It can design a procedure that will meet customer needs.
5. **Verify:** It can verify the design presentation and ability to meet customer needs.

8.7 SOFTWARE RELIABILITY

Software Reliability means **Operational reliability**. It described as the capability of a system or component to accomplish its required functions under static conditions for a specific period.

Software reliability is also defined as the prospect that a software system fulfils its assigned task in a given atmosphere for a predefined number of input cases, assuming that the hardware and the input are free of error.

Software Reliability is a necessary connect of software quality, composed with functionality, usability, performance, serviceability, capability, install ability, maintainability, and documentation. It is hard to achieve because the complexity of software turn to be high. While any system with a high degree of complexity, containing software, will be hard to reach a certain level of consistency, system developers tend to push complexity into the software layer, with the speedy growth of system size and ease of doing so by advancement the software.

8.8 THE ISO 9000 QUALITY STANDARDS

ISO (International Standards Organization) is a group or consortium of 63 countries established to plan and fosters standardization. ISO declared its 9000 series of standards in 1987. It serves as a reference for the contract between independent parties. The ISO 9000 standard determines the guidelines for maintaining a quality system. The ISO standard mainly addresses operational methods and organizational methods such as responsibilities, reporting, etc. ISO 9000 defines a set of guidelines for the production process and is not directly concerned about the product itself.

8.8.1-Types of ISO 9000 Quality Standards

The ISO 9000 series of standards is based on the hypothesis that if a proper stage is followed for production, then good quality products are bound to follow spontaneously. The types of industries to which the various ISO standards apply are as follows.

1. **ISO 9001:** This standard relates to the organizations involved in design, development, production, and servicing of goods. This is the standard that applies to most software development organizations.
2. **ISO 9002:** This standard applies to those organizations which do not design products but are only involved in the production. Therefore, ISO 9002 does not apply to software development organizations.
3. **ISO 9003:** This standard applies to organizations that are involved only in the installation and testing of the products. For example, Gas companies.

8.8.2 Steps to get ISO 9000 Certification:

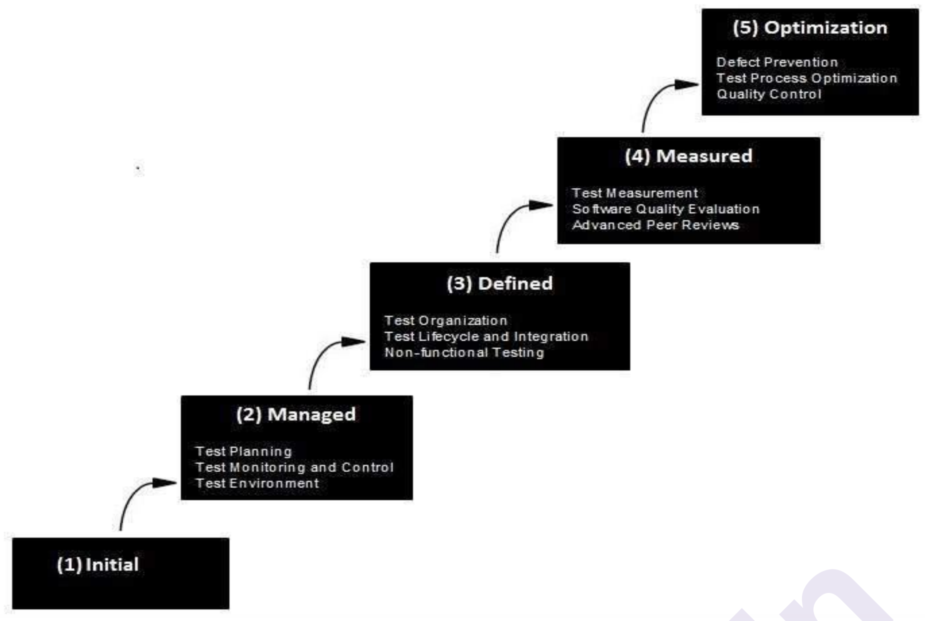
An organization decides to obtain ISO 9000 certification applies to ISO registrar office for registration. The process involves of the following stages:

Application -> Pre-Assessment -> Document review and Adequacy of Audit -> Compliance Audit -> Registration-> Continued Inspection.

8.9 CAPABILITY MATURITY MODEL

The Software Engineering Institute (SEI) Capability Maturity Model (CMM) states an increasing strings of levels of a software development business. The higher the level, the improved the software development process, therefore reaching each level is a costly and time-consuming method.

8.9.1 Levels of CMM



- **Level 1: Initial** - The software process is considered as unpredictable, and irregularly even disordered. Defined processes and standard practices that exist are unrestricted during a crunch. Success of the organization majorly be determined by on an individual effort. The heroes finally move on to other organizations taking their prosperity of knowledge or lessons learnt with them.

- **Level 2: Repeatable** - This level of Software Development Organization has a basic and steady project management procedures to track cost, schedule, and functionality. The process is in place to repeat the earlier achievements on projects with similar applications.

- **Level 3: Defined** - The software process for both management and engineering actions are documented, standardized, and integrated into a usual software process for the entire organization and all projects crosswise the organization use an approved, custom-made version of the organization's typical software process for developing, testing and maintaining the application.

- **Level 4: Managed** - Management can efficiently control the software development effort using specific measurements. At this level, organization set a quantifiable quality objective for both software process and software maintenance.

- **Level Five: Optimizing** - The Main characteristic of this level is fixing on continually improving process performance through both incremental and inventive technological improvements.

SUMMARY

Software Quality Assurance (SQA) is a set of activities for certifying quality in software engineering procedures. It ensures that developed software happens and fulfils with the defined or standardized quality provisions. SQA is an ongoing process within the Software Development Life Cycle (SDLC) that regularly checks the developed software to confirm it meets the anticipated quality measures.

LIST OF REFERENCES AND BIBLIOGRAPHY AND FURTHER READING:

- https://www.javatpoint.com/six-sigma
- https://www.computersprofessor.com/2017/09/sqa-tasks-goals-attributes-and-metrics.html

MODEL QUESTIONS

- What is the purpose of Software Quality Assurance?
- What are the elements of a SQA process?
- What are some common SQA activities and tasks?
- What is the role of metrics in SQA?
- What is the purpose of software quality standards (e.g., ISO 9001)?
- What is the Capability Maturity Model (CMM), and how is it used in SQA?
- What is the difference between verification and validation in the context of SQA?
- What is the purpose of software reliability engineering, and how does it relate to SQA?

❄❄❄❄❄❄❄

9

SOFTWARE TESTING

Unit Structure :

9.0 OBJECTIVES

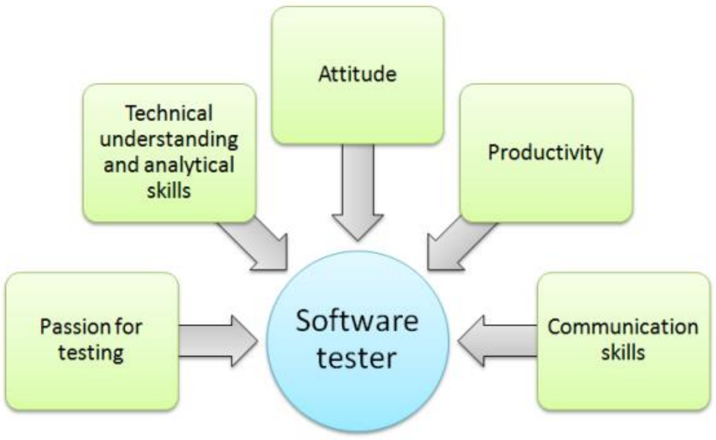
After going through this unit, students will be able to:

- Study fundamental concepts in software testing.
- Understand different levels and types of software testing.
- Understand the distinctions between software verification and software validation.

9.1 INTRODUCTION

Software testing is not anything but an art of examining software to ensure that its quality under test is in line with the requirement of the client. Software testing is carried out in an organized manner with the resolved of finding defects in a system. It is required for evaluating the system.

Software testing is now a very major and essential part of software development. Ideally, it is best to introduce software testing in every segment of software development life cycle. Actually, a common of software development time is now spent on testing.



9.2 VERIFICATION AND VALIDATION

Verification and validation are important concepts in software testing.

Verification refers to the process of calculating the software design and implementation to regulate whether it meets the listed requirements. Verification is a anticipatory process that aims to identify any defects and errors early in the software development lifecycle, before they become more tough and costly to fix. Verification activities include activities such as code assessments, design reviews, walkthroughs, and static analysis.

On the other hand, **Validation**, is the process of estimating the software during or at the end of the development process to regulate whether it satisfies the specified requirements. Validation is a remedial process that aims to identify defects and errors that may have been hosted during the implementation phase. Different types of Validation activities include unit testing, integration testing, system testing, and acceptance testing.

It's important to note that verification and validation are balancing processes and both are essential to ensure software quality. Verification helps to identify possible defects and errors early in the development process, whereas validation helps to identify faults and errors that may have been introduced later in the method. Effective verification and validation help to confirm that software meets the specified requirements, is free of faults and errors, and is suitable for its intended purpose.

9.3 INTRODUCTION TO TESTING

Testing is an important part of the software development process. It is the process of calculating a software system or its components with the determined to identify any defects or errors and to consider its functionality. The primary goal of testing is to ensure that the software meets the specified requirements and works as anticipated.

The choice of testing methods and techniques will depend on the specific needs and purposes of the software project, as well as the development methodology being used.

Effective testing involves a thorough understanding of the software requirements, as well as the development and testing methods. It's also important to develop a well-designed test plan, comprising a clear definition of the testing objectives, testing methods, and estimated outcomes. Effective testing is essential to ensure software quality and to minimize the risk of defects and errors in the final product.

Testing can be performed with the use of automated testing tools or manually. Automated testing can increase the proficiency and accuracy of the testing process, but it also requires a important investment of time and resources to develop and maintain.

9.4 TESTING PRINCIPLES

There are some principles that form the base of effective software testing:

- **Early Testing:** Testing should start early in the software development lifecycle and continue all over the process. This helps to identify defects and errors timely, when they are easier and less costly to fix.
- **Defect Prevention:** The focus should be on defect prevention rather than defect detection. This can be achieved by using proven software development methodologies, following best practices for software design and coding, and performing regular code reviews and walkthroughs.
- **Testing Throughout the Development Life Cycle:** Testing should be performed at all stages of the development life cycle, from requirements gathering and design through to implementation, testing, and deployment.
- **Test Planning and Design:** A well-designed test plan is essential for effective testing. The test plan should include a clear definition of the testing objectives, testing methods, and expected outcomes.
- **Independent Testing:** Testing should be performed by an independent team or individuals to ensure objectivity and to minimize the risk of bias.
- **Test-Driven Development:** Tests should be developed and executed before the implementation of the software components. This helps to ensure that the software meets the specified requirements and reduces the risk of defects.
- **Automation:** Automated testing can be an effective way to increase the efficiency and accuracy of the testing process. However, it's important to use automation appropriately and not rely solely on automated testing methods.
- **Continuous Testing:** Testing should be an ongoing process, not a one-time event. Continuous testing helps to ensure that changes to the software are tested and validated throughout the development life cycle.

In summary, these principles provide a framework for effective software testing and help to ensure that the software meets the specified requirements, is free of defects and errors, and is fit for its intended purpose.

9.5 TESTING OBJECTIVES

The objectives of software testing can vary depending on the specific needs of a software project. However, some common testing objectives include:

- **Verifying requirements:** Ensure that the software meets the detailed requirements and works as intended.
- **Finding defects:** Identify and separate defects and errors in the software.
- **Improving quality:** Improve the overall quality of the software by recognizing and fixing defects and successful the design and implementation.
- **Increasing confidence:** Increase confidence in the software by providing evidence that it meets the specified requirements and works as intended.
- **Evaluating risk:** Evaluate the potential risk associated with the software, including the risk of defects and the risk of security vulnerabilities.
- **Demonstrating compliance:** Demonstrate compliance with regulatory and industry standards, such as ISO 9001 or PCI DSS.
- **Improving reliability:** Improve the reliability of the software by reducing the frequency and severity of defects and errors.
- **Improving performance:** Improve the performance of the software by identifying and fixing performance bottlenecks and optimizing resource utilization.
- **Supporting maintenance:** Support ongoing software maintenance by providing information about the software's behavior and performance.

In summary, the objectives of software testing are to ensure that the software meets the specified requirements, works as intended, and is of high quality. Testing also helps to minimize the risk of defects and errors, improve performance, and support ongoing software maintenance.

9.6 TEST ORACLES

It is a mechanism, different from the program itself, that can be used to test the accuracy of a program’s output for test cases. Conceptually, we can consider testing a process in which test cases are given for testing and the program under test. The output of the two then compares to determine whether the program behaves correctly for test cases. This is shown in figure.

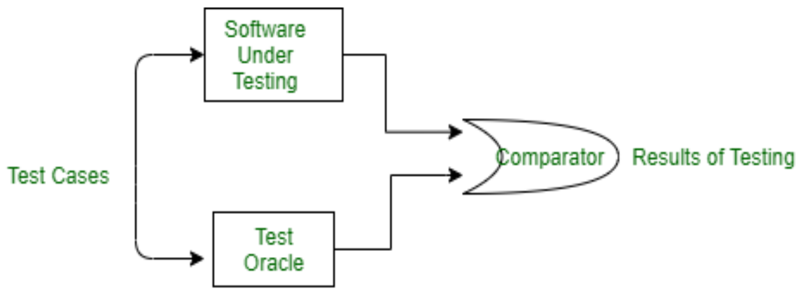


Figure - Testing and Test Oracles

Testing oracles are required for testing. Ideally, we want an automated oracle, which always gives the correct answer. However, often oracles are human beings, who mostly calculate by hand what the output of the program should be. As it is often very difficult to determine whether the behavior corresponds to the expected behavior, our "human deities" may make mistakes. Consequently, when there is a discrepancy, between the program and the result, we must verify the result produced by the oracle before declaring that there is a defect in the result.

The human oracles typically use the program's specifications to decide what the correct behavior of the program should be. To help oracle determine the correct behavior, it is important that the behavior of the system or component is explicitly specified and the specification itself be error-free. In other words, actually specify the true and correct behavior.

There are some systems where oracles are automatically generated from the specifications of programs or modules. With such oracles, we are assured that the output of the oracle conforms to the specifications. However, even this approach does not solve all our problems, as there is a possibility of errors in specifications.

As a result, a divine generated from the specifications will correct the result if the specifications are correct, and this specification will not be reliable in case of errors. In addition, systems that generate oracles from specifications require formal specifications, which are often not generated during design.

9.7 LEVELS OF TESTING

Testing can be divided into several levels, each of which serves a specific purpose and focuses on different aspects of the software. It is an important process in software development that helps ensure the quality and trustworthiness of a software product. Some most common levels of testing are as follows:

- **Unit Testing:** This is the first level of testing and involves testing individual components or units of code to ensure that each one functions as proposed. Unit tests are typically automated and are performed by developers.
- **Integration Testing:** This level of testing focuses on testing the interactions between different components or units of code. Integration testing helps to identify any issues that may arise from the integration of individual components.
- **System Testing:** System testing focuses on testing the entire software system as a whole, to ensure that it meets the specified requirements and behaves as expected. This level of testing may include functional testing, performance testing, and security testing.
- **User Acceptance Testing (UAT):** User Acceptance Testing is the final stage of testing, in which the software is tested by end-users or customers. The purpose of UAT is to ensure that the software meets the business requirements and satisfies the needs of the customers.
- **Performance Testing:** Performance testing is a type of testing that focuses on measuring the performance and scalability of a software system under different conditions, such as heavy load or high traffic. The goal of performance testing is to identify and resolve performance bottlenecks and ensure that the system can meet the expected performance requirements.

- **Security Testing:** Security testing is a type of testing that focuses on identifying and mitigating security vulnerabilities and threats in a software system. This level of testing includes vulnerability scans, penetration testing, and security assessments.

These levels of testing can be performed at different times during the software development lifecycle, and the exact testing process will depend on the specific requirements and constraints of the software project. However, it is generally recommended to perform testing at each level to ensure the quality and reliability of the final product.

9.8 WHITE-BOX TESTING/STRUCTURAL TESTING

Structural testing also known as White box testing, or code-based testing, is a type of software testing that emphases on the internal structure and design of a software program. It involves testing the individual components, functions, and modules of the code, as well as their collaborations with each other.

The goal of white box testing is to identify and correct any errors, bugs, or other issues in the code, and to ensure that it meets the specified requirements and design specifications. White box testing is often performed by developers and requires a detailed understanding of the code and how it works.

During white box testing, the tester has access to the source code and can test it at a low level, such as checking for proper syntax, data flow, and control flow. This type of testing is also used to validate the implementation of algorithms and data structures, as well as to test error handling and exception management.

White box testing complements other types of testing, such as black box testing and gray box testing, and is typically performed early in the software development lifecycle, before the software is released to the end-users. It is an important part of the software development process, as it helps to identify and resolve problems in the code, and ensures that the software is of high quality and reliable.

9.9 FUNCTIONAL/BLACK-BOX TESTING

Functional testing, also known as black box testing, is a type of software testing that focuses on verifying that the software meets the functional requirements and behaves as expected. Unlike white box testing, which focuses on the internal structure of the code, functional testing is performed from the perspective of an end-user, and does not require access to the source code.

The goal of functional testing is to validate the functionality of the software, including its inputs, outputs, and behavior. This type of testing focuses on testing the software's features and functions, and verifying that they work as intended.

Functional testing typically involves creating test cases and test scenarios that simulate real-world scenarios and interactions with the software. This can include manual testing, automated testing, or a combination of both.

Black box testing is performed at different stages of the software development lifecycle, and can be used to test the software as a whole, or individual components and functions. This type of testing is essential for ensuring that the software meets the user requirements and behaves as expected, and can help identify and resolve issues early in the development process, before the software is released to end-users.

In summary, functional testing is a crucial part of the software development process, and helps to ensure the quality and reliability of the software product. By performing functional testing, developers can validate that the software meets the specified requirements and behaves as expected, and can identify and resolve issues before the software is released to the end-users.

9.10 TEST PLAN

A test plan is a document that outlines the testing strategy, approach, and resources for a software project. It provides a roadmap for testing activities and helps ensure that the testing process is consistent, comprehensive, and aligned with the project requirements.

A typical test plan includes the following information:

- Introduction: A brief overview of the purpose and scope of the test plan.
- Objectives: The objectives of the testing process, such as verifying that the software meets the functional requirements, verifying the quality and reliability of the software, and identifying any issues or defects in the software.
- Scope: The scope of the testing process, including the components and functions that will be tested, and any areas or functionality that will not be tested.
- Test approach: The approach and methodology for testing, including the types of testing that will be performed (e.g., functional testing, performance testing, security testing, etc.), the testing tools and techniques that will be used, and the testing schedule.
- Test environment: The specifications and details of the testing environment, including the hardware, software, and network configurations, and the test data that will be used.
- Test cases: The test cases that will be used to verify the functionality and behavior of the software, including the steps, inputs, expected results, and pass/fail criteria.

- Test schedule: The testing schedule, including the start and end dates, the testing milestones, and the responsibilities of the testing team.
- Test resources: The resources required for testing, including the testing tools, personnel, and budget.
- Risks and assumptions: A description of the risks associated with the testing process and any assumptions that have been made.
- Approval: The approval process and sign-off criteria for the test plan.

The test plan is an important document that helps to ensure that the testing process is well-planned, well-organized, and consistent with the project requirements. It serves as a reference for the testing team and stakeholders, and helps to ensure that the testing process is completed on time, within budget, and with high quality.

9.11 TEST-CASE DESIGN

Test case design is the process of creating a set of tests to validate that the software functions as intended. Test cases are used to verify the functionality and behavior of the software, and to identify any issues or defects in the software.

Test case design involves several steps:

- **Identify requirements**: Start by identifying the functional requirements for the software and understanding what the software is expected to do.
- **Determine test conditions:** Based on the requirements, determine the conditions under which the software will be tested, such as different inputs, scenarios, and edge cases.
- **Design test cases:** Based on the test conditions, design test cases that will verify the functionality and behavior of the software. A test case should include a clear and concise description of the test steps, inputs, expected results, and pass/fail criteria.
- **Prioritize test cases:** Prioritize the test cases based on the risk and impact of each test. High-priority test cases should be designed and executed first, as they are more likely to uncover critical issues and defects.
- **Execute test cases:** Execute the test cases to verify the functionality and behavior of the software. Document the results and any issues or defects that are identified.
- **Update test cases:** Update the test cases based on the results of the testing, and make any necessary changes to the software. Repeat the testing process until all the test cases have been executed and the software meets the specified requirements.

Test case design is an iterative process that requires careful planning and organization. It is important to design test cases that are comprehensive, effective, and efficient, and to prioritize the test cases based on the risk and impact of each test. A well-designed set of test cases helps to ensure that the software functions as intended and is of high quality and reliability.

SUMMARY

- Software testing is required to check the reliability of the software
- Software testing ensures that the system is free from any bug that can cause any kind of failure
- Software testing ensures that the product is in line with the requirement of the client
- It is required to make sure that the final product is user friendly
- At the end software is developed by a team of human developers all having different viewpoints and approach. Even the smartest person has the tendency to make an error. It is not possible to create software with zero defects without incorporating software testing in the development cycle.
- No matter how well the software design looks on paper, once the development starts and you start testing the product you will find lots of defects in the design.

You cannot achieve software quality without software testing. Even if testers are not involved in actual coding, they should work closely with developers to improve the quality of the code. For best results it is important that software testing and coding should go hand in hand.

LIST OF REFERENCES AND BIBLIOGRAPHY AND FURTHER READING

- "Software Testing: A Craftsman's Approach" by Paul Jorgensen
- "Effective Software Testing: 50 Specific Ways to Improve Your Testing Process" by Elfriede Dustin, Thom Garrett, and Bernie Gauf
- "Introduction to Software Testing" by Paul Ammann and Jeff Offutt
- "Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design" by James A. Whittaker
- "Software Testing: An ISTQB-BCS Certified Tester Foundation Guide" by Rex Black, et al.

MODEL QUESTIONS

- What is software testing and why is it important?
- What are the different types of software testing?
- What is the difference between white box testing and black box testing?
- What is the purpose of test case design?
- What is the difference between functional testing and non-functional testing?
- What is the importance of test planning in software testing?
- What is the difference between verification and validation in software testing?
- What is the purpose of test automation and why is it important?
- What is the difference between bug and defect in software testing?
- What is the difference between static testing and dynamic testing?
- What is the difference between acceptance testing and user acceptance testing?
- What is the importance of test documentation in software testing?

❄❄❄❄❄❄❄