SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES (AUTONOMOUS)

MCA DEPARTMENT

LECTURE NOTES

COURSE : SOFTWARE ENGINEERING

YEAR / BRANCH : II MCA

REGULATION : R24

"Software is a great combination between artistry and engineering."

— Bill Gates

PREPARED BY :

Mrs. G. GEETHA,

Assistant Professor

II MCA – I SEMESTER

COURSE CODE: 24MCA212 CREDITS: 4 SOFTWARE ENGINEERING L-T-P: 3-1-0

UNIT - I : INTRODUCTION **Lecture Hrs:10**

The Evolving role of Software - Changing nature of Software - Legacy Software - Software myths. A layered technology- A Process Framework- CMMI- Process assessment - Personal and team Process Models.

UNIT - II : PROCESS MODELS **Lecture Hrs:12**

The waterfall model- Incremental process models- Evolutionary process models- Specialized Process Models- Agile process - Agile process Model: Extreme programming.

UNIT - III : SOFTWARE REQUIREMENTS AND SYSTEM MODELS **Lecture Hrs:12**

Functional and non-functional requirements- User requirements- System requirements- Interface specification- The software requirements document-Feasibility studies- Requirements elicitation and analysis-Requirements validation- Requirements management. Context Models- Behavioral models- Data models-structured methods.

UNIT - IV: DESIGN ENGINEERING& ARCHITECTURE, TESTING STRATEGIES **Lecture Hrs:12**

Design process and Design quality- Design concepts- the design model - Creating an architectural design: software architecture- Data design- Architectural styles and patterns- Architectural Design. A strategic approach to software testing- Test strategies for conventional Software - Validation testing-System testing-The art of debugging.

UNIT - V: TESTING TACTICS, SOFTWARE MEASUREMENT AND ESTIMATION **Lecture Hrs:12**

Software testing fundamentals - White-Box testing- Basis path testing- Control structure Testing- Black box testing. Size oriented metrics- Function oriented metrics- Metrics for software quality- Empirical Estimation Models: - Quality Management: Software quality assurance- Formal Technical Reviews.

TEXT BOOKS:

1. Software Engineering, A practitioner's Approach, 7/e, 2009, Roger S Pressman, Tata McGraw-Hill International Edition .

2. Software Engineering, 8/e, 2006, Ian Sommerville, Pearson Education, India.

REFERENCE BOOKS:

1. Fundamentals of Software Engineering, 2/e, 2005, Rajib Mall , Prentice Hall Inc, India.
2. Software Engineering: A Precise Approach , 1/e, 2010, Pankaj Jalote , Wiley, India.
3. Software Engineering: A Primer , 1/e, 2008, Waman S Jawadekar , Tata McGraw Hill , India.
4. Software Engineering - Principles and Practices ,1/e, Deepak Jain , Oxford University Press.

COURSE OUTCOMES: *On successful completion of this course, students will be able to:*		POs related to COs
CO1	**Demonstrate** the processes of software development.	PO1
CO2	**Analyze** the customer business requirements and choose the appropriate Process model for the project.	PO1,PO2, PO5,PO8
CO3	**Build** the prototype for Software business case and analyze the requirements of software project.	PO1,PO2,PO5, PO8
CO4	**Design** the System based on Architectural styles and Design patterns. .	PO1,PO2, PO3,PO5, PO8
CO5	**Design** test cases and Define metrics for standardization and assuring quality standards. .	PO1,PO2,PO3, PO3,PO8

CO-PO MAPPING(DETAILED; HIGH:3; MEDIUM:2; LOW:1)

Course	**POs** **COs**	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**
C302: Software Engineering	**C302.1**	**3**	**-**	**-**	**-**	**-**	**-**	**-**	**-**
	C302.2	**3**	**3**	**-**	**-**	**3**		**-**	**3**
	C302.3	**3**	**3**	**-**	**-**	**3**	**-**	**-**	**3**
	C302.4	**3**	**3**	**3**	**-**	**3**	**-**	**-**	**3**
	C302.5	**3**	**3**	**3**		**3**	**-**	**-**	**3**
	C302	**3**	**3**	**3**		**3**			**3**

UNIT – I

INTRODUCTION

"The function of good software is to make the complex appear to be simple."
— Grady Booch

Unit Overview

This unit introduces the **fundamentals of Software Engineering processes and technologies**. It explains how software has evolved over time, the changing nature of software systems, and the challenges of maintaining legacy software. The unit also discusses common software myths, layered technology in software engineering, process frameworks, process maturity models such as CMMI, process assessment techniques, and personal and team process models used to improve software quality and productivity.

Objectives of the Unit

After studying this unit, students should be able to:

- Understand the evolving role and changing nature of software.
- Explain the concept and characteristics of legacy software.
- Identify common software myths and their effects on software development.
- Describe software engineering as a layered technology.
- Understand the components of a software process framework.
- Explain the concept and levels of CMMI.
- Understand the need for software process assessment.
- Describe Personal Process Models (PSP) and Team Process Models (TSP).

Learning Outcomes

At the end of this unit, students will be able to:

- Explain the importance of software in modern society.
- Differentiate between modern software and legacy software.
- Identify misconceptions related to software development.
- Describe the layers of software engineering technology.
- Explain the activities involved in a software process framework.
- Discuss the importance of CMMI in process improvement.
- Evaluate software processes using assessment methods.
- Apply personal and team process models to improve software quality.

Importance of Studying this Unit

Studying this unit is important because:

- It provides a strong foundation in software engineering principles.
- It helps understand how software development processes improve quality.
- It teaches methods for managing and maintaining old software systems.
- It develops skills for individual and team-based software development.
- It prepares students for software industry practices and professional development.
- t introduces internationally accepted process improvement models like CMMI.

Key Concepts

1. The Evolving Role of Software

Software has evolved from simple programs to complex systems that control businesses, communication, healthcare, education, and entertainment.

Examples: Mobile apps, Cloud computing, Artificial intelligence systems.

2. Changing Nature of Software

Modern software is:

- Network-centric
- Web-based
- Mobile and cloud-enabled
- Continuously evolving
- Data-driven and intelligent

3. Legacy Software

Legacy software refers to old software systems that are still in use because they are critical to organizations.

Examples: Banking systems, government databases.

Problems:

- Difficult to maintain
- Outdated technology
- High maintenance cost

4. Software Myths

Software myths are false beliefs about software development.

Customer Myth:

"Requirements can be changed at any time without affecting cost."

Management Myth:

"If the project is late, add more programmers."

Practitioner Myth:

"Once the program works, the job is finished."

5. Software Engineering as a Layered Technology

Software engineering consists of four layers:

1. Quality Focus
2. Process
3. Methods
4. Tools

6. Process Framework

A process framework provides the structure for software development activities.

Framework Activities:

1. Communication
2. Planning
3. Modeling
4. Construction
5. Deployment

7. CMMI (Capability Maturity Model Integration)

CMMI is a process improvement model that helps organizations improve software development processes.

CMMI Levels:

1. Initial
2. Managed
3. Defined
4. Quantitatively Managed
5. Optimizing

8. Process Assessment

Process assessment evaluates the effectiveness and maturity of software processes.

Benefits:

- Improves quality
- Reduces risks
- Increases productivity
- Identifies weaknesses

9. Personal Process Models (PSP)

PSP helps individual software engineers improve their work by:

- Planning tasks
- Measuring performance
- Reducing defects
- Improving productivity

10. Team Process Models (TSP)

TSP helps teams:

- Plan projects effectively
- Improve communication
- Track progress
- Produce high-quality software

Unit I : Introduction

Definition: Software Engineering

Software engineering is a discipline that systematically applies engineering principles and best practices to the **planning, design, development, testing, deployment, and maintenance** of software systems. Its objective is to create high-quality, reliable, and scalable software solutions that meet user needs within constraints like time and budget.

Key Characteristics of Software

Unlike hardware, software must be engineered rather than traditionally manufactured. According to Pressman, it has four defining qualities:

Instructions: Computer programs that, when executed, provide desired features and performance.

Data Structures: Information that enables the programs to adequately manipulate and process data.

Documentation: Descriptive information in hard copy or digital formats that describes the operation and use of the programs.

No "Wear and Tear": Theoretically, software does not wear out like physical hardware; instead, it deteriorates over time due to unaddressed defects or the need for continuous updates and modifications.

Core Concepts of Software Engineering

Software engineering relies on a structured, layered approach to solve complex problems:

- **The Layered Technology:** Software engineering functions as a layered technology built on a **Quality Focus**, which drives the process, methods, and tools.
- **Process Framework:** A set of repeatable activities that provide stability, control, and organization to development teams.
- **Abstraction and Modeling:** Creating simplified representations of complex systems (like architecture diagrams) to understand requirements before writing code.
- **Modularity:** Dividing a large software system into smaller, independent modules to make development, testing, and debugging manageable.

1. Communication & Requirements Engineering

- **Definition:** Defining the exact problem to solve by gathering requests from stakeholders.
- **Example Tasks:** Hosting workshops with business investors, interviewing drivers, and surveying potential passengers.
- **Ride-Sharing App Example:** The team discovers that passengers need a way to track their driver in real-time, drivers need an in-app map, and both require secure digital payment processing.

2. Planning & Estimation

- **Definition:** Mapping out the timeline, budget, resource allocation, and potential project risks.
- **Example Tasks:** Setting a six-month deadline, assigning a budget, and planning for data privacy regulations.
- **Ride-Sharing App Example:** The project manager schedules the payment gateway integration for month three and flags GPS signal loss as a major technical risk that requires a backup tracking method.

3. Modeling & Architectural Design

- **Definition:** Creating a visual blueprint of the software architecture, database design, and user interface.
- **Example Tasks:** Drawing user interface sketches (wireframes) and designing how data flows between the user and the server.
- **Ride-Sharing App Example:** Designers draw the screen layouts for the passenger and driver apps. Software architects decide to use a cloud-based server database to instantly match available drivers with nearby passengers.

4. Construction (Coding & Testing)

- **Definition:** Writing the actual program code and running rigorous tests to eliminate errors.
- **Example Tasks:** Writing backend APIs, developing the front-end interface, and executing automated test scripts.
- **Ride-Sharing App Example:** Developers write Swift code for iOS and Kotlin code for Android. QA engineers test the app by simulating 10,000 simultaneous ride requests to ensure the system does not crash.

5. Deployment & Maintenance

- **Definition:** Launching the product to the live market and continuously updating it based on feedback.
- **Example Tasks:** Publishing apps to app stores, monitoring server speeds, and patching security vulnerabilities.
- **Ride-Sharing App Example:** The app launches in a major city. When users report that the app drains their phone battery too quickly during long rides, the engineering team releases a software update to optimize GPS battery usage.

The Evolving role of Software

Software is the engine that drives **modern businesses and industries**, and its role in software engineering continues to evolve at a rapid pace. Originally, software was seen as a product of engineering, a static collection of instructions for computers. Today, **it serves a dual purpose: both a product that delivers computing potential and a dynamic vehicle for delivering services and managing information.**

This shift has driven the evolution of software engineering from an **ad-hoc, craft-based approach to a more structured, disciplined, and agile methodology that prioritizes continuous iteration, user feedback, and faster delivery cycles.**

Here's how this evolution plays out with concrete examples:

1. From standalone programs to interconnected systems

Early software: Simple, single-purpose programs like those used for scientific calculations or basic data processing.

Example: A program calculating the line of a rocket, running on a single computer.

Modern software: Complex, interconnected systems that interact with numerous other applications and devices, often leveraging cloud-native architectures and DevOps practices.

Example: An e-commerce platform relies on numerous **microservices running on a cloud-native** platform . This allows for independent development, deployment, and scaling of features **like product catalog management, shopping cart functionality, payment processing, and recommendation engines**, all working together to deliver a seamless customer experience.

2. From ad-hoc development to iterative and agile methodologies

Early development: Often characterized by ad-hoc programming, with limited emphasis on planning, documentation, and formal testing.

Modern development: Embraces iterative and agile methodologies focusing on flexibility, customer collaboration, and frequent delivery of working software.

Example: Instead of a lengthy, upfront design process, an Agile team developing a new mobile banking app will release small, working versions (Minimum Viable Products) with core functionalities, gathering user feedback, and iteratively enhancing the application based on those insights. This could involve rolling out new features like fingerprint login or contactless payments in short sprints, ensuring rapid adaptation to market needs.

3. From manual tasks to automation and AI integration

Traditional software engineering: Many tasks, including coding, testing, and deployment, were often manual and time-consuming.

Modern software engineering: Leverages automation and AI/ML tools throughout the development lifecycle, enhancing efficiency and accuracy.

Example: AI-powered tools like GitHub , Copilot can assist developers by suggesting code snippets, automating repetitive coding tasks, and helping them write cleaner, more effective code. In testing, AI-driven systems generate adaptive test cases and prioritize critical tests, while automated **DevOps processes streamline deployment and monitoring in CI/CD pipelines.**

4. From software product to a vehicle for delivering value

Traditional software: Focused on the core functionality of the software itself.

Modern software: Acts as a vehicle for delivering value across various industries, impacting how people work, communicate, and live.

Example: Software in healthcare facilitates electronic health records, telemedicine, and AI-powered diagnostic tools, enhancing patient care and operational efficiency. In finance, AI-driven solutions are used for fraud detection, risk management, and personalized investment advice.

5. From specialized developers to cross-functional teams and citizen developers

Traditional development: Roles were more siloed, with developers focused on specific coding tasks.

Modern software engineering: Emphasizes cross-functional teams that collaborate across the entire software development lifecycle.

Example: DevOps teams integrate development and operations, fostering shared responsibility and **continuous feedback loops**. This collaborative culture, combined with **agile practices**, allows for faster and more reliable **software deployments**. Additionally, the rise of low-code/no-code platforms empowers "**citizen developers**" (non-technical individuals) to create applications **tailored to their specific needs**, reducing the reliance on specialized IT departments.

Software is both a product and a vehicle for delivering a product.

THE CHANGING NATURE OF SOFTWARE:

System software: System software is a collection's 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.

Application software

Application software consists of **standalone programs** that solve 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.** In addition to conventional **data processing applications**, application software is used to control **business functions in real-time** (e.g., point-of-sale transaction processing, real-time manufacturing process control)

Embedded software:

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.

Embedded software can perform limited and esoteric functions (e.g ., **keypad control for a microwave oven)** or provide significant function and control capability **(e.g., digital functions in an automobile such as fuel control, dashboard displays etc.).**

Product-Line software:

Designed to provide a **specific** capability for use by many **different customers**, product-line software can focus on a **limited and esoteric marketplace** (e.g., **inventory control products)** or address mass consumer markets (e.g., **word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications)**

Web-applications:

"WebApps," span 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**. However, as **e-commerce and B2B applications** grow in importance, WebApps are evolving into sophisticated computing environments that not only provide **standalone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.**

Artificial intelligence software: AI software makes **use of non numerical 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.**

Open source: A growing trend that results in **distribution of source code for systems applications** (e.g., operating systems, database, and development environments) so that customers can make local modifications.

The challenge for software engineers is to build source code that is self-descriptive, but, more importantly, to develop techniques that will enable **both customers and developers** to know **what changes have been made and how those changes marked themselves** within the software.

Legacy software

Legacy software refers to outdated computer systems, software applications, or technologies that are still in use within an organization, despite the availability of newer, more efficient alternatives. These systems often continue to function because they still perform required tasks, and replacing or updating them can be costly or risky. However, they can cause significant challenges for businesses seeking to modernize their IT infrastructure and remain competitive.

Key concepts: Legacy Software

1. **Outdated technology**

- **Concept:** Legacy systems often rely on programming languages or hardware that are no longer widely used or supported.
- **Example:** Many financial institutions continue to use core banking systems built with COBOL, a programming language from the 1950s. This poses challenges in finding developers and maintaining these systems.

2. Business-critical functionality

- **Concept**: These systems are often vital for core business operations and are deeply integrated into daily workflows.
- **Example:** The IRS Individual Master File (IMF) system, established in the 1960s using Assembly Language and COBOL, is still foundational for tax collection in the U.S. Its age presents challenges in scalability and modernization.

3. Maintenance challenges and lack of expertise

- **Concept:** Maintaining legacy systems can be difficult due to outdated codebases and a scarcity of skilled personnel proficient in older technologies.
- **Example:** As COBOL programmers retire, it becomes increasingly difficult and expensive to maintain COBOL-based systems that handle critical functions in banking and government sectors.

4. **Scalability limitations**

- **Concept:** Many legacy systems struggle to adapt and scale to meet growing data volumes, increasing users, or evolving business needs.
- **Example**: The US Ski and Snowboard team had a split legacy data management system that caused operational issues and slowed down user experience. The team turned to modernization to address these limitations.

5. **Security vulnerabilities**

- **Concept:** Legacy systems may lack modern security features and updates, making them vulnerable to cyber attacks and data breaches.

- **Example:** Legacy medical devices pose patient safety and data security risks due to their reliance on outdated software and lack of robust security updates.

6. **Integration difficulties**
   - **Concept:** Legacy systems often struggle to communicate and integrate with newer systems and technologies due to compatibility issues.
   - **Example:** Old banking systems may not integrate well with modern mobile applications, hindering customer experience and efficiency.

Benefits of legacy systems
Despite the challenges, organizations often choose to retain legacy systems due to their:

- **Proven reliability**: They have a demonstrated track record of stable performance over time.
- **Tailored functionality**: Many were custom-built to perfectly meet specific business processes and workflows.
- **Data preservation**: They contain vast amounts of valuable historical data, crucial for compliance and analysis.
- **Cost avoidance (short-term):** Replacing them can be a significant investment, involving the cost of new hardware, software, and migration efforts.

Modernization approaches
Organizations can address the challenges of legacy software through various modernization approaches:

- **Rehosting** (Lift-and-Shift): Moving the application to a new infrastructure like the cloud with minimal changes.
- **Replatforming**: Migrating the application to a modern platform while preserving its core functionality and data.
- **Refactoring**: Restructuring the existing code to improve performance, scalability, and maintainability without altering the core functionality.
- **Reengineering**: Redesigning and rebuilding the application from the ground up using modern technologies and best practices.
- **Replacing with new systems**: Completely replacing the legacy system with a modern solution.

Software myths

Software myths are **false beliefs or misconceptions** about software development and its processes. These can impact all aspects of software projects, including management, development, customer expectations, timelines, resources, and even the final product quality. By recognizing and dispelling these myths, organizations can create a strong development process aligned with realistic goals and methodologies.

Here are some common software myths with examples:

1. Management myths

- **Myth**: Adding more developers to a delayed project will automatically speed up its completion.
- **Reality**: Adding new developers can actually delay a project further, as existing team members need to spend time on training and on boarding. This is known as Brooks's Law: "Adding manpower to a late software project makes it later".

2. Customer myths

- **Myth:** General requirements are sufficient for developers to create a satisfactory product.
- **Reality**: Unclear requirements can lead to increased costs and delays. Detailed and exact documentation is essential for developers to understand the client's vision and build the desired product.
- **Myth:** Software is easy to change or modify after it has been developed.
- **Reality:** Changes after development can be complex and expensive, potentially requiring significant rework.

3. Developer myths

- **Myth**: More code means better software.
- **Reality:** High-quality software is determined by factors like **clean architecture and efficient algorithms**, not the **amount of code**. Concise and maintainable code is often preferred for readability and ease of debugging.
- **Myth**: Software can be completely bug-free.
- **Reality:** Achieving completely bug-free software is practically impossible due to the complexity of modern systems. The focus should be on building reliable and secure software.

Layered Technology

Software Engineering is a fully layered technology, to develop software we need to go from one layer to another. All the layers are connected and each layer demands the fulfillment of the previous layer.

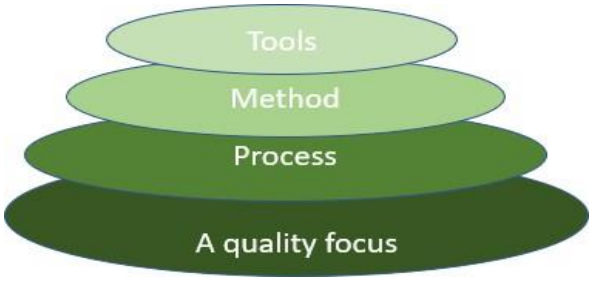


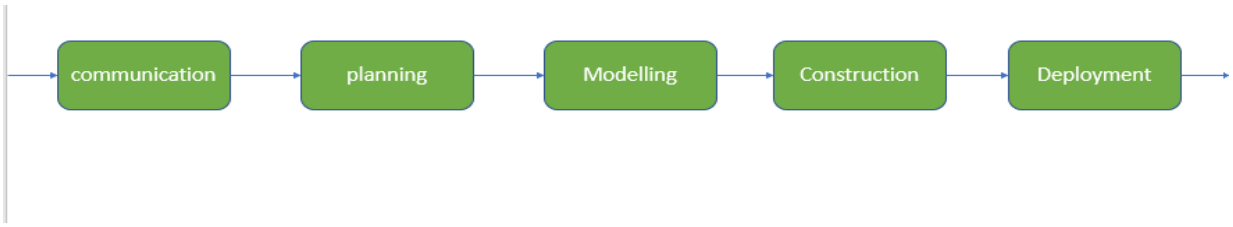
Fig: The diagram shows the layers of software development

Layered technology is divided into four parts:

1. A quality focus: It defines the continuous process improvement principles of software. It provides integrity that means providing security to the software so that data can be accessed by only an authorized person, no outsider can access the data. It also focuses on maintainability and usability.

2. Process: It is the foundation or base layer of software engineering. It is key that binds all the layers together which enables the development of software before the deadline or on time.

Process defines a framework that must be established for the effective delivery of software engineering technology. The software process covers all the activities, actions, and tasks required to be carried out for software development.



Process activities are listed below:-

- **Communication:** It is the first and foremost thing for the development of software. Communication is necessary to know the actual demand of the client.
- **Planning:** It basically means drawing a map for reduced the complication of development.
- **Modeling:** In this process, a model is created according to the client for better understanding.
- **Construction:** It includes the coding and testing of the problem.
- **Deployment:-** It includes the delivery of software to the client for evaluation and feedback.

3. Method: During the process of software development the answers to all "how-to-do" questions are given by method. It has the information of all the tasks which **includes communication, requirement analysis, design modeling, program construction, testing, and support**.

4. Tools: Software engineering tools provide a self-operating system for processes and methods. Tools are integrated which means information created by one tool can be used by another.

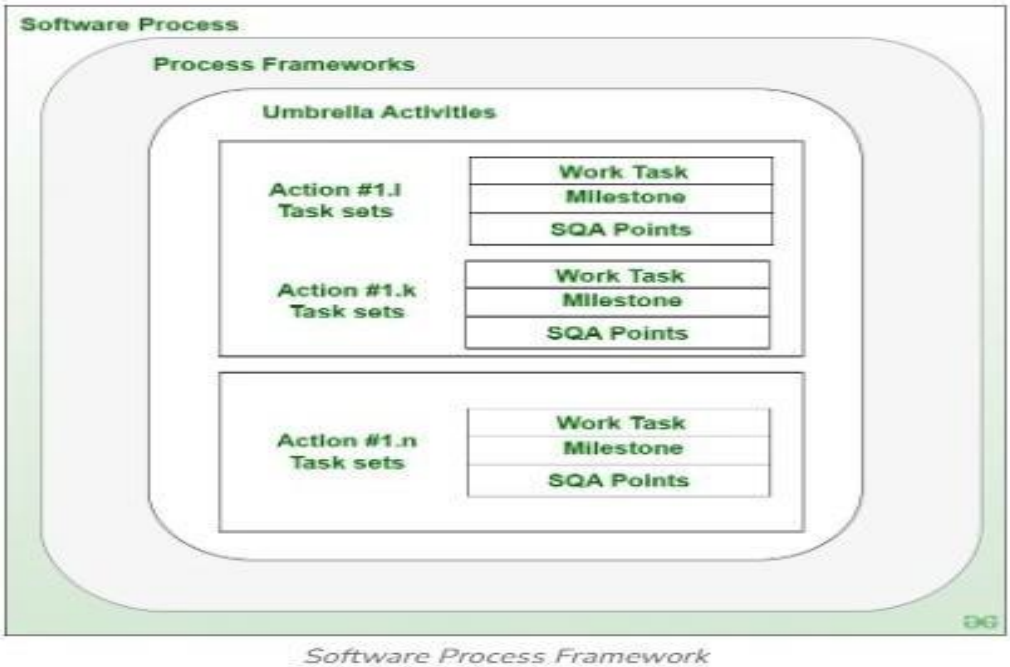
Software Process Framework

A **Software Process Framework** is a structured approach that defines the steps, tasks, and activities involved in software development. This framework serves as a **foundation for software engineering**, guiding the development team through various stages to ensure a **systematic and efficient process**. A Software Process Framework helps in project planning, risk management, and quality assurance by detailing the chronological order of actions.

It includes task sets, umbrella activities, and process framework activities, all essential for a **successful software development lifecycle**. Utilizing a well-defined Software Process Framework enhances productivity, consistency, and the overall quality of the software product.

Since it serves as a foundation for them, it is utilized in most applications. Task sets, umbrella activities, and process framework activities all define the characteristics of the software development process. Software Process includes:

1. **Tasks:** They focus on a small, specific objective.
2. **Action:** It is a set of tasks that produce a major work product.
3. **Activities:** Activities are groups of related tasks and actions for a major objective.



Software Process Framework

The Software process framework is required for representing common process activities. Five framework activities are described in a process framework for software engineering. Communication, planning, modeling,

construction, and deployment are all examples of framework activities. Each engineering action defined by a framework activity comprises a list of needed work outputs, project milestones, and software quality assurance (SQA) points. Let's explain each:

Software Process Framework Activities

1. Communication:

Definition**: Communication involves gathering requirements from customers and stakeholders to determine the system's objectives and the software's requirements.**

Activities:

Requirement Gathering: Engaging with consumers and stakeholders through meetings, interviews, and surveys to understand their needs and expectations.

Objective Setting: Clearly defining what the system should achieve based on the gathered requirements.

Explanation: Effective communication is essential to understand what the users need from the software. This phase ensures that all stakeholders are on the same page regarding the goals and requirements of the system.

2. Planning:

Definition**:** Planning involves establishing an engineering work plan, describing technical risks, listing resource requirements, and defining a work schedule.

Activities:

Work Plan: Creating a detailed plan that outlines the tasks and activities needed to develop the software.

Risk Assessment: Identifying potential technical risks and planning how to mitigate them.

Resource Allocation: Determining the resources (time, personnel, tools) required for the project.

Schedule Definition: Setting a timeline for completing different phases of the project.

Explanation: Planning helps in organizing the project and setting clear expectations. It ensures that the development team has a roadmap to follow and that potential challenges are anticipated and managed.

3. Modeling:

Definition**: Modeling involves creating architectural models and designs to better understand the problem and work towards the best solution.**

Activities:

Analysis of Requirements: Breaking down the gathered requirements to understand what the system needs to do.

Design: Creating architectural and detailed designs that outline how the software will be structured and how it will function.

Explanation: Modeling translates requirements into a visual and structured representation of the system. It helps in identifying the best design approach and serves as a blueprint for development.

4. Construction:

Definition**: Construction involves creating code, testing the system, fixing bugs, and confirming that all criteria are met.**

Activities:

Code Generation: Writing the actual code based on the design models.

Testing: Running tests to ensure the software works as intended, identifying and fixing bugs.

Explanation: This phase is where the actual software is built. Testing is crucial to ensure that the code is error-free and that the software meets all specified requirements.

5. Deployment:

Definition**: Deployment involves presenting the completed or partially completed product to customers for evaluation and feedback, then making necessary modifications based on their input.**

Activities:

Product Release: Delivering the software to users, either as a full release or in stages.

Feedback Collection: Gathering feedback from users about their experience with the software.

Product Improvement: Making changes and improvements based on user feedback to enhance the product.

Umbrella Activities

Umbrella Activities that take place during a software development process for improved project management and tracking.

1. **Software project tracking and control:** This is an activity in which the team can assess progress and take corrective action to maintain the schedule. Take action to keep the project on time by comparing the project's progress against the plan.
2. **Risk management:** The risks that may affect project outcomes or quality can be analyzed. Analyze potential risks that may have an impact on the software product's quality and outcome.
3. **Software quality assurance:** These are activities required to maintain software quality. Perform actions to ensure the product's quality.
4. **Formal technical reviews:** It is required to assess engineering work products to uncover and remove errors before they propagate to the next activity. At each level of the process, errors are evaluated and fixed.
5. **Software configuration management:** Managing of configuration process when any change in the software occurs.
6. **Work product preparation and production:** The activities to create models, documents, logs, forms, and lists are carried out.
7. **Reusability management:** It defines criteria for work product reuse. Reusable work items should be backed up, and reusable software components should be achieved.
8. **Measurement:** In this activity, the process can be defined and collected. Also, project and product measures are used to assist the software team in delivering the required software.

Capability Maturity Model Integration (CMMI)

The Capability Maturity Model Integration (CMMI) is an advanced framework designed to improve and integrate processes across various disciplines such as software engineering, systems engineering, and people management. It builds on the principles of the original CMM, enabling organizations to enhance their processes systematically. CMMI helps organizations fulfill customer needs, create value for investors, and improve product quality and market growth. It offers two representations, staged and continuous, to guide organizations in their process improvement efforts.

Objectives of CMMI

1. Fulfilling customer needs and expectations.
2. Value creation for investors/stockholders.
3. Market growth is increased.
4. Improved quality of products and services.
5. Enhanced reputation in Industry.

CMMI Representation - Staged and Continuous

A representation allows an organization to pursue a different set of improvement objectives. There are two representations for CMMI :

Staged Representation:

- uses a pre-defined set of process areas to define improvement path.
- provides a sequence of improvements, where each part in the sequence serves as a foundation for the next.
- an improved path is defined by maturity level.
- maturity level describes the maturity of processes in organization.
- Staged CMMI representation allows comparison between different organizations for multiple maturity levels.

Continuous Representation :

- Allows selection of specific process areas.
- Uses capability levels that measures improvement of an individual process area.
- Continuous CMMI representation allows comparison between different organizations on a process-area-by-process-area basis.
- Allows organizations to select processes which require more improvement.
- In this representation, order of improvement of various processes can be selected which allows the organizations to meet their objectives and eliminate risks.

CMMI Model - Maturity Levels

In CMMI with staged representation, there are five maturity levels described as follows :

Maturity level 1 : Initial

- processes are poorly managed or controlled.
- unpredictable outcomes of processes involved.
- ad hoc and chaotic approach used.
- No KPAs (Key Process Areas) defined.
- Lowest quality and highest risk.

Maturity level 2 : Managed

- Requirements are managed.
- Processes are planned and controlled.
- Projects are managed and implemented according to their documented plans.
- This risk involved is lower than Initial level, but still exists.
- Quality is better than Initial level.

Maturity level 3 : Defined

- Processes are well characterized and described using standards, proper procedures, and methods, tools, etc.
- Medium quality and medium risk involved.
- Focus is process standardization.

Maturity level 4 : Quantitatively managed

- Quantitative objectives for process performance and quality are set.
- Quantitative objectives are based on customer requirements, organization needs, etc.
- Process performance measures are analyzed quantitatively.
- Higher quality of processes is achieved.
- lower risk

Maturity level 5 : Optimizing

- Continuous improvement in processes and their performance.
- Improvement has to be both incremental and innovative.
- Highest quality of processes.
- Lowest risk in processes and their performance.

CMMI Model - Capability Levels

A capability level includes relevant specific and generic practices for a specific process area that can improve the organization's processes associated with that process area. For CMMI models with continuous representation, there are six capability levels as described below :

Capability level 0 : Incomplete

- Incomplete process - partially or not performed.
- one or more specific goals of process area are not met.
- No generic goals are specified for this level.
- this capability level is same as maturity level 1.

Capability level 1: Performed

- Process performance may not be stable.
- Objectives of quality, cost and schedule may not be met.
- A capability level 1 process is expected to perform all specific and generic practices for this level.
- Only a start-step for process improvement.

Capability level 2: Managed

- Process is planned, monitored and controlled.
- Managing the process by ensuring that objectives are achieved.

- Objectives are both model and other including cost, quality, schedule.
- Actively managing processing with the help of metrics.

Capability level 3: Defined

- A defined process is managed and meets the organization's set of guidelines and standards.
- Focus is process standardization.

Capability level 4 : Quantitatively Managed

- Process is controlled using statistical and quantitative techniques.
- Process performance and quality is understood in statistical terms and metrics.
- Quantitative objectives for process quality and performance are established.

Capability level 5 : Optimizing

- Focuses on continually improving process performance.
- Performance is improved in both ways - incremental and innovation.
- Emphasizes on studying the performance results across the organization to ensure that common causes or issues are identified and fixed.

Software Process Assessment

Software process assessment (SPA) is a **systematic evaluation of an organization's software development processes aimed at identifying strengths, weaknesses, and areas for improvement**. It helps organizations understand and optimize their operations.

Key elements of software process assessment

- **Defining the Scope and Objectives**: Determine which processes will be assessed, which could include the entire software development lifecycle or specific areas.
- **Selecting an Assessment Framework**: Choose a framework like CMMI, ISO/IEC 15504 (SPICE), or ISO 9001 that aligns with organizational goals and resources.
- **Planning the Assessment**: Create a detailed plan with timelines and resources.
- **Conducting the Assessment**: Gather data through interviews, observations, and document reviews.
- **Analyzing the Results**: Identify strengths and weaknesses and evaluate processes against the chosen framework.
- **Developing Recommendations**: Provide actionable recommendations for improvement.
- **Implementing Improvements**: Prioritize and implement the recommended changes, potentially using Agile methodologies.
- **Verifying and Re-assessing**: Verify the effectiveness of changes and conduct re-assessments.

Example 1: Implementing CMMI to improve software quality

A company facing project delays used CMMI. After an assessment showing chaotic processes (Level 1), they moved to Level 2 by establishing basic project management. Progressing to Level 3 involved integrating processes and implementing improvements like continuous integration, which led to improved software quality, reduced costs, and faster development.

Example 2: Using process assessment to address software development challenges

A team with frequent bugs due to last-minute requirement changes used process assessment to reveal inadequate collaboration and insufficient detail in user stories. They adopted early testing and Agile workflows. Process assessment can also help manage technical debt by identifying areas for refactoring and incorporating practices like code reviews and automated testing.

Personal and team Process models:

Personal Software Process (PSP)

The personal software process (PSP) is focused on individuals to improve their performance. The PSP is an individual process, and it is a bottom-up approach to software process improvement. The PSP is a prescriptive process, it is a more mature methodology with a well-defined set of tools and techniques.

Key Features of Personal Software Process (PSP)

Following are the key features of the Personal Software Process (PSP):

- **Process-focused:** PSP is a process-focused methodology that emphasizes the importance of following a disciplined approach to software development.
- **Personalized:** PSP is personalized to an individual's skill level, experience, and work habits. It recognizes that individuals have different strengths and weaknesses, and tailors the process to meet their specific needs.
- **Metrics-driven:** PSP is metrics-driven, meaning that it emphasizes the collection and analysis of data to measure progress and identify areas for improvement.
- **Incremental:** PSP is incremental, meaning that it breaks down the development process into smaller, more manageable pieces that can be completed in a step-by-step fashion.
- **Quality-focused:** PSP is quality-focused, meaning that it emphasizes the importance of producing high-quality software that meets user requirements and is free of defects.

Advantages of Personal Software Process (PSP)

Improved productivity: PSP provides a structured approach to software development that can help individuals improve their productivity by breaking down the development process into smaller, more manageable steps.

Improved quality: PSP emphasizes the importance of producing high-quality software that meets user requirements and is free of defects. By collecting and analyzing data throughout the development process, individuals can identify and eliminate sources of errors and improve the quality of their work.

Personalized approach: PSP is tailored to an individual's skill level, experience, and work habits, which can help individuals work more efficiently and effectively.

Improved estimation: PSP emphasizes the importance of accurate estimation, which can help individuals plan and execute projects more effectively.

Continuous improvement: PSP promotes a culture of continuous improvement, which can help individuals learn from past experiences and apply that knowledge to future projects.

Disadvantages of Personal Software Process (PSP)

Following are the Disadvantages of Personal Software Process (PSP):

Time-consuming: PSP can be time-consuming, particularly when individuals are first learning the methodology and need to collect and analyze data throughout the development process.

Complex: PSP can be complex, particularly for individuals who are not familiar with software engineering concepts or who have limited experience in software development.

Heavy documentation: PSP requires a significant amount of documentation throughout the development process, which can be burdensome for some individuals.

Limited to individual use: PSP is designed for individual use, which means that it may not be suitable for team-based software development projects.

Team Software Process

Team Software Process (TSP) is a team-based process. TSP focuses on team productivity. Basically, it is a top-down approach. The TSP is an adaptive process, and process management methodology.

Key Features of Team Software Process (TSP):

Team-focused: TSP is team-focused, meaning that it emphasizes the importance of collaboration and communication among team members throughout the software development process.

Process-driven: TSP is process-driven, meaning that it provides a structured approach to software development that emphasizes the importance of following a disciplined process.

Metrics-driven: TSP is metrics-driven, meaning that it emphasizes the collection and analysis of data to measure progress, identify areas for improvement, and make data-driven decisions.

Incremental: TSP is incremental, meaning that it breaks down the development process into smaller, more manageable pieces that can be completed in a step-by-step fashion.

Quality-focused: TSP is quality-focused, meaning that it emphasizes the importance of producing high-quality software that meets user requirements and is free of defects.

Feedback-oriented: TSP is feedback-oriented, meaning that it emphasizes the importance of receiving feedback from peers, mentors, and other stakeholders to identify areas for improvement.

Advantages of Team Software Process (TSP):

Following are the key advantage of the Team Software Process (TSP):

Improved productivity: TSP provides a structured approach to software development that can help teams improve their productivity by breaking down the development process into smaller, more manageable steps.

Improved quality: TSP emphasizes the importance of producing high-quality software that meets user requirements and is free of defects. By collecting and analyzing data throughout the development process, teams can identify and eliminate sources of errors and improve the quality of their work.

Team collaboration: TSP promotes team collaboration, which can help teams work more efficiently and effectively by leveraging the skills and expertise of all team members.

Improved estimation: TSP emphasizes the importance of accurate estimation, which can help teams plan and execute projects more effectively.

Continuous improvement: TSP promotes a culture of continuous improvement, which can help teams learn from past experiences and apply that knowledge to future projects.

Disadvantages of Team Software Process (TSP):

Time-consuming: TSP can be time-consuming, particularly when teams are first learning the methodology and need to collect and analyze data throughout the development process.

Complex: TSP can be complex, particularly for teams that are not familiar with software engineering concepts or who have limited experience in software development.

Heavy documentation: TSP requires a significant amount of documentation throughout the development process, which can be burdensome for some teams.

Requires discipline: TSP requires teams to follow a disciplined approach to software development, which can be challenging for some teams who prefer a more flexible approach.

Cost: TSP can be costly to implement, particularly if teams need to invest in training or software tools to support the methodology.

Key Highlights

- **The Evolving Role of Software**: Acts simultaneously as a **product** (transforms data) and a **vehicle** (controls other systems).
- **Changing Nature of Software**: Transitioned from classic standalone programs to **ubiquitous Web/Cloud ecosystems**, mobile apps, and real-time AI networks.
- **Legacy Software**: Decades-old software supporting **critical business workflows** that are complex, high-risk, and costly to replace.
- **Software Myths**: False, misleading assumptions about software management, shifting customer requirements, and practitioner development habits

Process Frameworks & Engineering Infrastructure

- **A Layered Technology**: Built on four fundamental structural levels: **Quality focus** (bedrock) ->**Process** -> **Methods** -> **Tools**.
- **A Process Framework**: Consists of 5 **Framework Activities** (Communication, Planning, Modeling, Construction, Deployment) managed by **Umbrella Activities**
- **CMMI**: A 5-level organizational maturity scale: **Initial** -> **Managed** -> **Defined** -> **Quantitatively Managed** -> **Optimizing**.
- **Process Assessment**: Formal diagnostic procedures (e.g., SCAMPI appraisals) used to measure an organization's actual software process maturity level.
- **Personal & Team Process Models**: **PSP** teaches individual engineers to track time and defects; **TSP** organizes them into self-directed, high-performance units.

Textbooks

1. Software Engineering: A Practitioner's Approach by Roger S. Pressman.
2. Software Engineering by Ian Sommerville.

Reference Books

1. Software Engineering: A Precise Approach by Pankaj Jalote.
2. Software Engineering: Principles and Practice by Waman S. Jawadekar.
3. Fundamentals of Software Engineering by Rajib Mall.
4. A Discipline for Software Engineering by Watts S. Humphrey.

Case Study

1. Legacy Software

- **Example: A Bank's Core ATM System**
- **The Scenario:** A major bank still runs its main savings account transaction system on code written in the 1980s using COBOL.
- **The Problem:** The code works perfectly for calculating interest, but it is incredibly difficult to modify to support modern smartphone banking apps because the original programmers have retired and there is no user manual.

2. Software Myths

- **Example: The Delayed Hospital Booking System**
- **The Scenario:** A hospital software project is two months behind schedule. The manager believes a common *Management Myth* and says, *"If we hire 5 more programmers right now, we will catch up and finish on time."*
- **The Reality:** The new programmers actually slow the project down further because the existing developers have to stop their own work to train them.

3. A Layered Technology

- **Example: Building a Ride-Sharing App (Like Uber)**
- **How the Layers Work:**
    - **Quality Layer (Bedrock):** The company sets a strict rule that the app must never crash during a ride.
    - **Process Layer:** They set up weekly meetings and timelines to plan features like map tracking and payments.
    - **Methods Layer:** Engineers write the technical code step-by-step to calculate the shortest path using GPS data.
    - **Tools Layer:** Developers use automated software testing tools to check the code instantly every time it changes.

4. Personal and Team Process Models (PSP / TSP)

- **Example: A College Student Project vs. A Professional Team**
- **How it Works:**
    - **Personal Process (PSP):** An individual programmer keeps a private logbook tracking exactly how many minutes they spend writing code and how many typos they make. They use this data to accurately predict how long their next assignment will take.
    - **Team Process (TSP):** A group of 5 developers meets together to vote on their own project milestones, distribute work evenly based on their personal records, and check each other's code to ensure zero bugs are sent to customers.

Question Number	Questions	Bloom's Level:
Unit – I INTRODUCTION		
PART – A (2 Marks)		
1	What is the dual role of software?	L1
2	List four distinct categories of software based on its changing nature.	L1
3	Define legacy software. [Bloom's Level: L1 - Remembering]	L1
4	Give two examples of a management myth in software engineering.	L1
5	Name the four layers of software engineering technology.	L1
6	List the five generic activities in a process framework.	L1
7	What are the two types of representations used in CMMI?	L1
8	Define process assessment.	L1
9	State the primary objective of the Personal Software Process (PSP).	L1
10	State the primary objective of the Team Software Process (TSP).	L1
11	Why does legacy software need to change instead of being discarded?	L2
12	Distinguish between a framework activity and an umbrella activity.	L2
13	What is the main difference between PSP and TSP?	L2
14	If a project is late, why does adding more programmers make it later?	L3
15	Why is "Quality Focus" placed at the bottom layer of software technology?	L4
PART – B(10 Marks)		
1	Explain the evolving role and changing nature of modern software categories.	L2
2	Discuss legacy software, why it deteriorates, and the strategies used to maintain it.	L2
3	Classify and explain Management, Customer, and Practitioner myths along with their realities.	L2
4	Describe Software Engineering as a Layered Technology with a neat structural diagram.	L2
5	Explain the architecture of a generic Process Framework including all its core components.	L2
6	Explain the five maturity levels of the CMMI Staged Representation model.	L2
7	Discuss the process assessment procedure and contrast the SCAMPI and SPICE approaches.	L2
8	Explain the framework phases, goals, and metrics of the Personal Software Process (PSP).	L2
9	Describe the Team Software Process (TSP) framework and detail its typical project launch cycle.	L2
10	Apply the CMMI Continuous Representation schema to improve specific weak process areas in a firm.	L3
11	Analyze how conflicting expectations between Customer Myths and Practitioner Myths damage a project.	L4

UNIT – II

PROCESS MODELS

"An incremental process gives you early visibility. It is much easier to steer a moving vehicle than one that is sitting on blocks in a garage."

—— Craig Larman

Unit Overview

This unit introduces the major software process models used in software engineering. It explains how different development models help organize software projects from planning to maintenance. The unit begins with the traditional Waterfall Model and progresses to modern Agile methodologies, including Extreme Programming (XP). It also discusses Incremental, Evolutionary, and Specialized Process Models, enabling students to understand when and why each model is used.

Objectives of the Unit

After studying this unit, students should be able to:

- Understand the concept of software process models.
- Explain the phases of the Waterfall Model.
- Describe Incremental and Evolutionary development models.
- Differentiate Specialized Process Models.
- Understand the principles of Agile software development.
- Explain the practices of Extreme Programming (XP).
- Select an appropriate process model for different software projects.

Learning Outcomes

After completing this unit, students will be able to:

- Describe different software process models.
- Compare traditional and Agile software development approaches.
- Explain the characteristics of Incremental and Evolutionary models.
- Identify suitable applications of Specialized Process Models.
- Apply Agile principles in software development.
- CO6: Explain the workflow and practices of Extreme Programming (XP).

Importance of Studying this Unit

- Provides the foundation of software development methodologies.
- Helps in selecting the right process model for software projects.
- Improves software quality and project management.
- Reduces development risks and project failures.
- Supports faster software delivery through Agile practices.
- Enhances teamwork and customer involvement.
- Prepares students for industry-standard software development practices.

Key Concepts

1. Software Process Model

A structured framework that defines the activities required to develop software.

2. Waterfall Model

A sequential software development model where each phase is completed before the next begins.

3. Incremental Process Model

Software is developed and delivered in small functional increments.

4. Evolutionary Process Model

Software evolves through repeated development and customer feedback.

5. Specialized Process Models

Process models designed for specific application domains such as Component-Based Development, Formal Methods, Aspect-Oriented Software Development, and Concurrent Development.

6. Agile Process

An iterative and customer-focused software development approach emphasizing flexibility and rapid delivery.

7. Extreme Programming (XP)

An Agile process model that promotes frequent releases, continuous customer involvement, pair programming, test-driven development, refactoring, and continuous integration.

Definition : Software Process Model

A software process model is a **structured representation of the activities of the software development process**. During the development of software, **various steps that are important for the successful development of the project** are taken and if we **structured them according to the proper order in a model** then it is called a **software process model**. The software process model includes various activities such as steps like planning, designing, implementation, defining tasks, setting up milestones, roles, and responsibilities, etc.

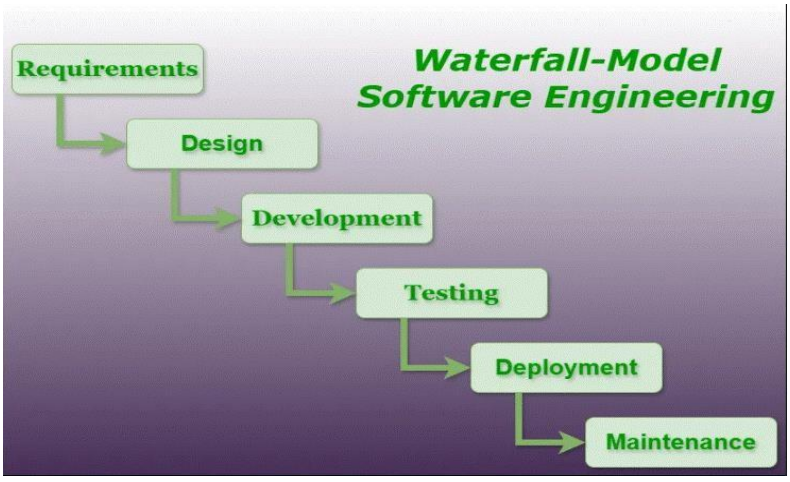
Waterfall Model

The waterfall model is a Software Development Model **used in** the **context of large, complex projects,** typically in the field of **information technology**. It is characterized by a **structured, sequential approach** to **Project Management** and **Software Development**.

The Waterfall Model is **useful in situations** where the **project requirements are well-defined** and the **project goals are clear**. It is often **used for large-scale projects with long timelines**, where there is little room for error and the project stakeholders need to have a high level of confidence in the outcome.

Phases of Waterfall Model

Classical Waterfall Model divides the life cycle into a **set of phases.** The development process can be considered as a sequential flow in the waterfall. The different sequential phases of the classical waterfall model are follow:



Requirements Analysis and Specification

Requirement Analysis and specification **phase aims to understand the exact requirements of the customer and document them properly.** This phase consists of two different activities.

1.1 Requirement Gathering and Analysis: Firstly **all the requirements regarding the software** are **gathered from the customer** and then the **gathered requirements are analyzed.**

The goal of the **analysis part is to remove incompleteness** (an incomplete requirement is one in which some parts of the actual requirements have been omitted) and **inconsistencies** (an inconsistent requirement is one in which some part of the requirement contradicts some other part).

1.2. Requirement Specification:

These **analyzed requirements are documented** in a software requirement specification (SRS) document. **SRS document** serves as a **contract between the development team and customers**. Any **future dispute between the customers and the developers can be settled** by examining the SRS document.

2. Design

The goal of this **Software Design Phase** is to convert the **requirements acquired in the SRS** into a format that **can be coded in a programming language**. It includes **high-level** and **detailed design as well as the overall software architecture**. A **Software Design Document** is used to document all of this effort (SDD).

High-Level Design (HLD): This phase focuses on outlining the broad structure of the system. It highlights the key components and how they interact with each other, giving a clear overview of the system's architecture.

Low-Level Design (LLD): Once the high-level design is in place, this phase zooms into the details. It breaks down each component into smaller parts and provides specifics about how each part will function, guiding the actual coding process.

3. Development

In the Development Phase software design is translated into source code using any suitable programming language. Thus each designed module is coded. The unit testing phase aims to check whether each module is working properly or not**.**

In this phase, developers begin writing the actual source code based on the designs created earlier.

The goal is **to transform the design into working code** using the most suitable programming languages.

Unit tests are often performed during this phase **to make sure that each component functions correctly on its own.**

4. Testing and Deployment

Testing:

Integration of different modules is undertaken soon after they have been coded and unit tested. Integration of **various modules** is **carried out incrementally over several steps**. During **each integration step, previously planned modules are added to the partially integrated system** and the **resultant system is tested**. Finally, after **all the modules** have been successfully **integrated and tested**, the full working system is obtained and **system testing** is carried out on this. System testing consists of **three different kinds of testing** activities as described below.

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

6. Deployment:

Once the software has been thoroughly tested, it's time **to deploy it to the customer or end-users.** This means making the software **ready and available for use**, often by **moving it to a live or staging environment.**

During this phase, we also **focus on helping users get comfortable with the software by providing training, setting up necessary environments, and ensuring everything is running smoothly.** The goal is to make sure the system works as **expected in real-world conditions and that users can start using it without any hitches.**

7. Maintenance

In **Maintenance Phase** is the most important phase of a software life cycle. The effort spent on maintenance is 60% of the total effort spent to develop a full software. There are three types of maintenance.

- **Corrective Maintenance:** This type of maintenance is **carried out to correct errors that were not discovered during the product development phase.**
- **Perfective Maintenance:** This type of maintenance is **carried out to enhance the functionalities of the system based on the customer's request.**
- **Adaptive Maintenance:** Adaptive maintenance is usually required for porting the software to work in a new environment such as working on a new computer platform or with a new operating system.

Features of Waterfall Model

- **Sequential Approach**: The waterfall model involves a sequential approach to software development, where **each phase of the project is completed before moving on to the next one.**
- **Document-Driven:** The waterfall model **depended on documentation to ensure that the project is well-defined and the project team is working towards a clear set of goals.**
- **Quality Control:** The waterfall model places a high **emphasis on quality control and testing at each phase of the project, to ensure that the final product meets the requirements and expectations** of the stakeholders.
- **Rigorous Planning**: The waterfall model involves a **careful planning process**, where the **project scope, timelines, and deliverables are carefully defined and monitored** throughout the project lifecycle.

Importance of Waterfall Model

- **Clarity and Simplicity:** The linear form of the Waterfall Model offers a simple and unambiguous foundation for project development.
- **Clearly Defined Phases:** The Waterfall Model phases each have **unique inputs and outputs, guaranteeing a planned development with clear checkpoints.**
- **Documentation:** A focus on thorough documentation helps with **software understanding, maintenance, and future growth.**

- **Stability in Requirements:** Suitable for projects when the requirements are **clear and stable, reducing modifications as the project progresses.**
- **Resource Optimization:** It encourages effective task-focused work **without continuously changing contexts by allocating resources according to project phases.**

Incremental Process Model

The Incremental Process Model is a method of software development where the **system is built step by step**. Instead of **delivering the whole system at once**, it is **developed** and **delivered in small parts** called **increments**. **Each increment builds upon** the **previous one by adding new functionality**, until the **complete system is finished**.

Key Characteristics of Incremental Process Model

- **Partial System Delivery:** The system is **developed and delivered in small**, manageable pieces. Each part **adds new features to the previous version**.
- **Early Functionality: Basic functionality** is available **early in the project**. This allows **users to start** using and **testing the system quickly**.
- **Customer Feedback Loop**: **Feedback is collected** after **each part is delivered**. This helps
- **improve** the **next version of the system**.
- **Flexible to Changes: Changes** or **new features** can be **added between increments**. This makes the model flexible to **evolving needs**.
- **Combination of Linear and Iterative Approaches:** Combines the structured approach of Waterfall with flexibility. Supports both planning and ongoing improvements**.**

Phases of the Incremental Model

The phases of Incremental model is divided into the four parts which is **Requirement**, **Design**, **Testing** and **Implementation** phases. In those phase, **the process continues until we got the expected output at the end.**

Phases of incremental model

Requirement Analysis

The first step in the Incremental Model is **understanding what the software needs** to do. The **team gathers the requirements** from **the product experts and clearly** defines the **system's functional needs**. This phase is important because it **sets the foundation for everything** else in the development process.

Design & Development

Next, the team focuses on **designing how the software will function** and **starts developing it**. They work on **adding new features** and **making sure the system works as expected**. The design and development steps go **hand-in-hand to build the functionality of the software.**

Testing

Once a feature is developed, it goes through testing. The testing phase **checks how the software performs, including both new and existing features.** The team **uses different testing methods** to make sure **everything is working correctly**.

Implementation

This phase involves **writing the final code based on the design and development steps.** After testing the functionality, the team **verify that everything is working as planned**. By the end of this phase, the product is **regularly improved and updated** until **it becomes the final working version**.

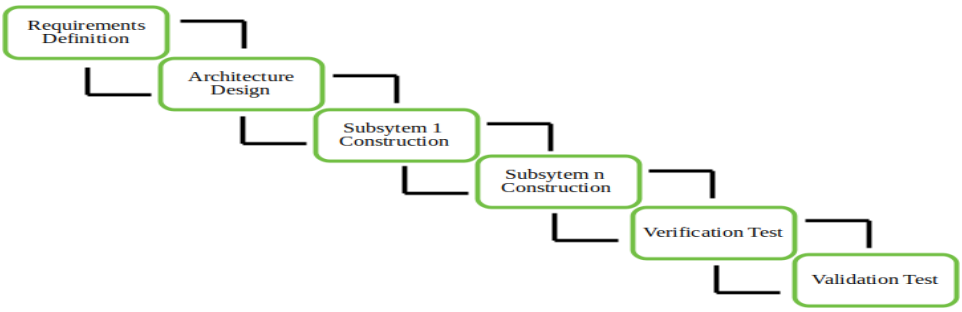
Types of Incremental Model

The Incremental Model has two main types, each offers different approaches to how software is developed in parts. Here are the two types:

Staged Delivery Model

The Staged Delivery Model **develops software in a sequence of planned stages**, where **each stage delivers a functional part of the system**. **Each release** brings the **product closer to completion,** allowing it to evolve regularly. Working **versions are delivered** at **regular intervals**, making **progress visible and manageable** throughout the **development process**.

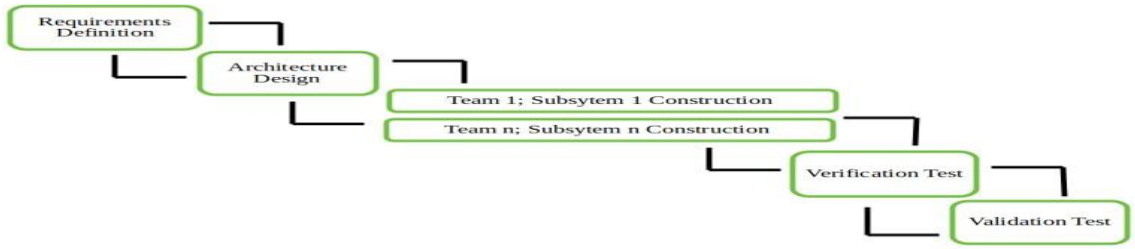
The diagram below shows this model :



Staged Delivery Model

Parallel Development Model

The Parallel Development Model **divides the system into multiple modules** that are **developed simultaneously at the same time by different teams**. By working on **separate components in parallel**, the **development process becomes faster** and more **efficient.** This approach **reduces overall project time** and allows **teams to focus on specific functionalities concurrently**. Given below is the diagram showing the model:



Parallel Development Model

Use Cases of Incremental Process Model

When the requirements are well-defined and clear:
Because increments can be planned and developed step-by-step with minimal requirement changes.

If the project has a long development timeline:

Incremental delivery helps manage complexity over time by breaking the project into smaller, manageable parts.

If the customer needs a quick product release:

You can deliver the most critical features early in the first increment, allowing the customer to start using the software sooner.

When you want to develop the most important features first:

This allows early feedback on key functionalities and better prioritization for subsequent increments.

Advantages of Incremental Process Model

The Incremental Model of software development builds a system in **small, manageable sections** (increments), making it a **good choice for many projects**. Below are the key advantages:

Faster Software Delivery

- **Initial working versions of the software** can be **delivered quickly**.
- Early delivery **increases customer satisfaction** and feedback opportunities.

Clear Understanding for Clients

- Clients get **to see parts of the system at each stage.**
- This visibility ensures that the **final product meets their expectations.**

Easy to Implement Changes

- Requirements can evolve, and **changes can be integrated in subsequent** increments.
- It supports **flexibility without heavily disrupting earlier stages.**

Effective Risk Management

- **Risks can be identified and handled early** due to the staged approach.
- **Each increment allows for testing and validation, reducing** the impact of unpredicted issues.

Flexible Criteria and Scope

- **Requirements** can be **adjusted without a major cost increase**.
- Better scope management helps keep the project aligned with business goals.

Cost-Effective

- Compared to models like the Waterfall, the incremental model is generally more cost-efficient.
- Budget is spread across stages, making it easier to manage finances**.**

Simpler Error Identification

- Since **development is done in parts**, it's **easier to pinpoint and fix errors within a specific increment**.

- Testing each module separately enhances **quality and reliability.**

Disadvantages of Incremental Process Model

Incremental Model comes with some limitations and challenges. Below are the key disadvantages:

Requires a Skilled Team and Proper Planning

- Successful implementation demands an experienced team.
- Poor planning or coordination can lead to confusion between increments.

Cost Can Increase Over Time

- **Due to repeated testing, redesign, and integration in every cycle,** the overall **project cost** may **rise.**
- **Continuous iteration** involves **added overhead**

Incomplete Requirement Gathering Can Cause Design Issues

- **If all requirements** are **not identified early**, the **system architecture** may **not support future needs.**
- Insufficient upfront design can **lead to rework** and architectural mismatches.

Lack of Smooth Flow Between Increments

- **Each iteration** may **function independently**, which can create inconsistencies.
- There might be integration challenges when **combining all the increments** into a integrated product.

High Effort to Fix Repeated Issues

- **A defect in one increment** may **exist in others**.
- Fixing the **same issue across multiple units** can be **time-consuming** and **resource-intensive.**

Evolutionary Process Model

The evolutionary model is based on the concept of making an initial product and then evolving the software product over time with iterative and incremental approaches with proper feedback. In this type of model, the product will go through several iterations and come up when the final product is built through multiple iterations. The development is carried out simultaneously with the feedback during the development. This model has a number of advantages such as customer involvement, taking feedback from the customer during development, and building the exact product that the user wants. Because of the

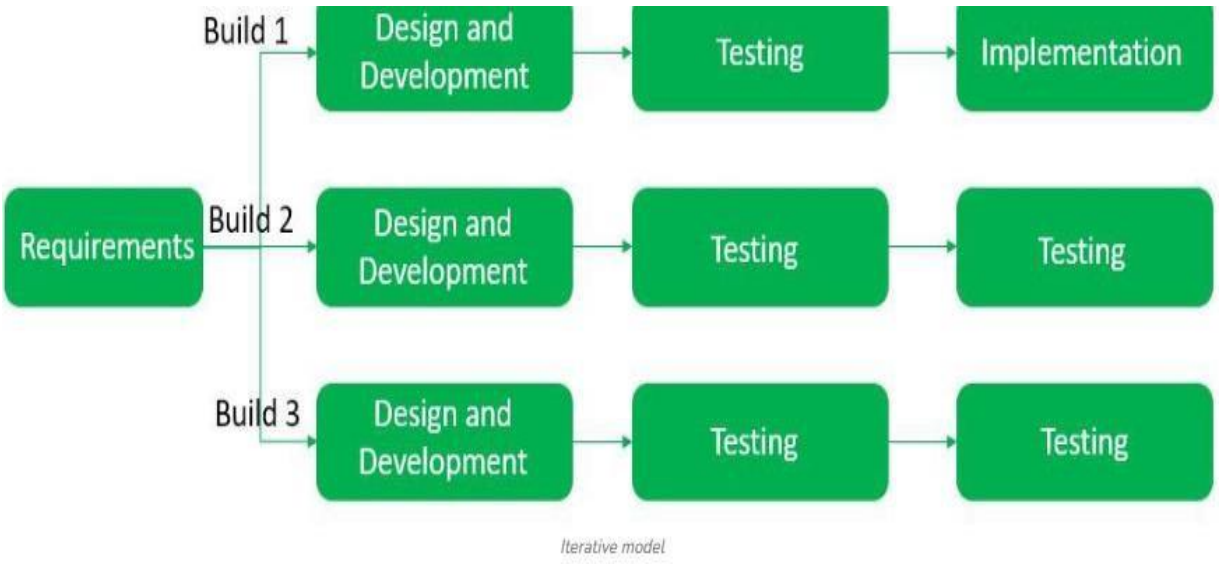
multiple iterations, the chances of errors get reduced and the reliability and efficiency will increase.

Types of Evolutionary Process Models

1. Iterative Model
2. Incremental Model
3. Spiral Model

1. Iterative Model

In the iterative model first, we take the initial requirements then we enhance the product over multiple iterations until the final product gets ready. In every iteration, some design modifications were made and some changes in functional requirements is added. The main idea behind this approach is **to build the final product through multiple iterations** that result in the **final product** being almost the same as the **user wants with fewer errors and the performance, and quality would be high.**



Iterative model

Iterative and Incremental development is a combination of both iterative design or iterative method and incremental build model for development. "During software development, more than one iteration of the software development cycle may be in progress at the same time." This process may be described as an "evolutionary acquisition" or "incremental build" approach."

In this incremental model, the whole requirement is divided into various builds. During each iteration, the development module goes through the requirements, design, implementation and

testing phases. Each subsequent release of the module adds function to the previous release. The process continues till the complete system is ready as per the requirement.

The key to a successful use of an iterative software development lifecycle is rigorous validation of requirements, and verification & testing of each version of the software against those requirements within each cycle of the model. As the software evolves through successive cycles, tests must be repeated and extended to verify each version of the software.

Advantages of the Iterative Model:

- **Early risk mitigation:** Potential problems can be identified and addressed early in the development process.
- **Adaptability to change:** The model allows for easier integration of changing requirements or feedback.
- **Improved quality:** Continuous feedback and testing lead to a more robust and refined product.
- **Faster time to market:** Working versions of the software are available in each iteration.
- **Increased stakeholder involvement:** Stakeholders can provide feedback throughout the development process.

Disadvantages of the Iterative Model:

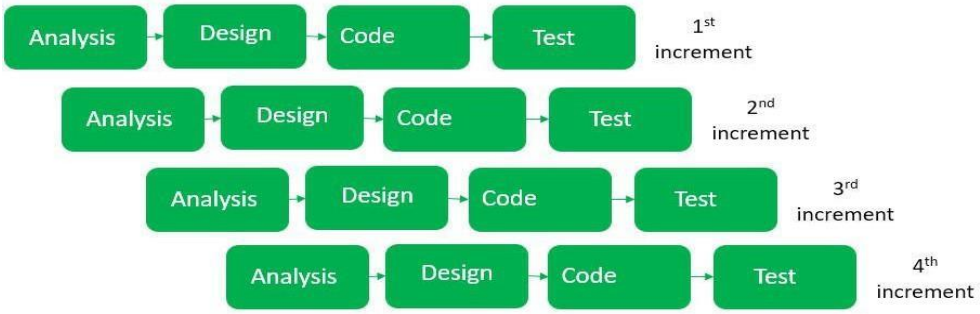
- **Increased complexity:** Managing iterations and ensuring consistency can be challenging.
- **Potential for scope creep:** Uncontrolled changes during iterations can lead to scope creep.
- **Higher initial costs:** Iterative development can be more resource-intensive.

Examples of Iterative Model:

- **Prototyping:** Creating and refining prototypes in iterative cycles.
- **Agile development methodologies:** Frameworks like Scrum and Kanban utilize iterative and incremental approaches.
- **Rational Unified Process (RUP):** RUP incorporates iterative and incremental development as a core principle.
- **Rapid Application Development (RAD):** RAD emphasizes rapid prototyping and iterative development.

2. Incremental Model

In the incremental model, we first build the project with basic features and then evolve the project in every iteration, it is mainly used for large projects. The first step is to gather the requirements and then perform analysis, design, code, and test and this process goes the same over and over again until our final project is ready.



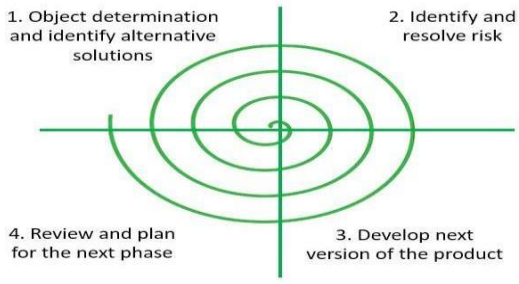
Incremental Model

3. Spiral Model

The Spiral Model is a **Software Development Life Cycle (SDLC)** model that provides a systematic and iterative approach to software development. In its diagrammatic representation, looks like a spiral with many loops. The exact number of loops of the spiral is unknown and can vary from project to project. Each loop of the spiral is called a **phase** of the software development process.

Some **Key Points** regarding the **Stages of a Spiral Model**:

1. The exact number of phases needed to develop the product can be varied by the project manager depending upon the project risks.
2. As the project manager dynamically determines the number of phases, the project manager has an important role in developing a product using the spiral model.
3. It is based on the idea of a spiral, with each iteration of the spiral representing a complete software development cycle, from requirements gathering and analysis to design, implementation, testing, and maintenance.



Spiral Model

Phases of the Spiral Model

The Spiral Model is a risk-driven model, meaning that the focus is on managing risk through multiple iterations of the software development process. Each phase of the Spiral Model is divided into four Quadrants:

1. Objectives Defined

In first phase of the spiral model we clarify what the project aims to achieve, including functional and non-functional requirements.

Requirements are gathered from the customers and the objectives are identified, elaborated, and analyzed at the start of every phase. Then alternative solutions possible for the phase are proposed in this quadrant.

2. Risk Analysis and Resolving

In the risk analysis phase, the risks associated with the project are identified and evaluated.

During the second quadrant, all the possible solutions are evaluated to select the best possible solution. Then the risks associated with that solution are identified and the risks are resolved using the best possible strategy. At the end of this quadrant, the Prototype is built for the best possible solution.

3. Develop the next version of the Product

During the third quadrant, the identified features are developed and verified through testing.

At the end of the third quadrant, the next version of the software is available.

In the evaluation phase, the software is evaluated to determine if it meets the customer's requirements and if it is of high quality.

4. Review and plan for the next Phase

In the fourth quadrant, the Customers evaluate the so-far developed version of the software. In the end, planning for the next phase is started.

The next iteration of the spiral begins with a new planning phase, based on the results of the evaluation.

The Spiral Model is often used for complex and large software development projects, as it allows for a more flexible and adaptable approach to **Software development**. It is also well-suited to projects with significant uncertainty or high levels of risk.

Advantages of the Evolutionary Process Model

1. During the development phase, the customer gives feedback regularly because the customer's requirement gets clearly specified.
2. After every iteration risk gets analyzed.

3. Suitable for big complex projects.
4. The first build gets delivered quickly as it used an iterative and incremental approach.
5. **Enhanced Flexibility:** The iterative nature of the model allows for continuous changes and refinements to be made, accommodating changing requirements effectively.
6. **Risk Reduction:** The model's emphasis on risk analysis during each iteration helps in identifying and mitigating potential issues early in the development process.
7. **Adaptable to Changes:** Since changes can be incorporated at the beginning of each iteration, it is well-suited for projects with evolving or uncertain requirements.
8. **Customer Collaboration:** Regular customer feedback throughout the development process ensures that the end product aligns more closely with the customer's needs and expectations.

Disadvantages of the Evolutionary Process Model

1. It is not suitable for small projects.
2. The complexity of the spiral model can be more than the other sequential models.
3. The cost of developing a product through a spiral model is high.
4. **Project Management Complexity:** The iterative nature of the model can make project management and tracking more complex compared to linear models.
5. **Resource Intensive:** The need for continuous iteration and customer feedback demands a higher level of resources, including time, personnel, and tools.
6. **Documentation Challenges:** Frequent changes and iterations can lead to challenges in maintaining accurate and up-to-date documentation.
7. **Potential Scope Creep:** The flexibility to accommodate changes can sometimes lead to an uncontrolled expansion of project scope, resulting in scope creep.
8. **Initial Planning Overhead:** The model's complexity requires a well-defined initial plan, and any deviations or adjustments can be time-consuming and costly.

Specialized process models

Specialized process models in software engineering are adaptations of traditional software development models that address particular project needs or challenges encountered with conventional methods. They are chosen when the project requires a specific approach due to factors such as size, complexity, technical hurdles, or features like security, reusability, or automation.

Here are some examples of these models with illustrative scenarios:

- **Component-Based Development (CBD) Model:**

  - Description: This model leverages pre-existing software components (like Commercial off-the-shelf software or COTS) to construct software, aiming for faster development and lower costs. It encourages modularity and reuse through a cycle of component identification, integration, customization, testing, deployment, and maintenance.

  - Example: A company developing a new Enterprise Resource Planning (ERP) system might opt for CBD, using pre-built components for common functions like accounting, human resources, or inventory management. This allows them to focus on developing unique business logic, reducing development time and effort. The reuse of these components also enhances the system's maintainability and scalability.

- **Formal Methods Model**:

  - **Description:** This model employs mathematical techniques and formal logic to specify, develop, and verify software systems, ensuring high reliability and correctness, particularly in critical applications. Formal methods use rigorous proofs and mathematical reasoning to ensure the system behaves as intended under all conditions, addressing issues like ambiguity, incompleteness, and inconsistency.

  - **Example**: Developing software for a nuclear reactor's control system, where errors can have catastrophic consequences, would benefit from formal methods. Engineers would use mathematical specifications to model the system's behavior and verify that it will respond correctly to any possible temperature input, including edge cases that might be missed by traditional testing methods.

- **Aspect-Oriented Software Development (AOSD) Model:**

  - Description: This model enhances modularity by separating crosscutting concerns, such as logging, security, or error handling, into independent modules called aspects. This separation improves code organization, reduces redundancy, and simplifies maintenance and updates.

  - Example: In a large banking application, a crosscutting concern like security might involve authentication, authorization, and data encryption. Using AOSD, a security aspect could encapsulate all security-related logic, which is then automatically applied across various modules of the application, such as user login, transaction processing, and data reporting. This makes the code base more organized and easier to manage.

Agile Process

Agile is a widely adopted methodology in software engineering that emphasizes an iterative, collaborative, and flexible approach to software development. It focuses on delivering working software frequently, incorporating customer feedback, and adapting to changing requirements. This differs from traditional, sequential approaches like the Waterfall model.

Here's a closer look at the agile process:

1. Core principles and values

Agile is guided by the Agile Manifesto, a document created in 2001 by a group of software developers. Its four core values are:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

These values are further supported by 12 principles that provide more concrete guidance.

2. Phases of the agile development lifecycle

The agile software development lifecycle (SDLC) typically includes six phases:

1. **Concept:** Defining the project vision, scope, and objectives.
2. **Inception**: Building the development team, designing the system architecture, and gathering detailed requirements.
3. **Iteration (Construction**): Developers work in short cycles (sprints) to build and refine the software, delivering working increments.
4. **Release**: Testing and quality assurance, user training, and deployment of the software.
5. **Maintenanc**e: Ongoing support, bug fixes, and feature enhancements based on feedback.
6. **Retirement**: Replacing the software with a new system or phasing it out when it becomes outdated.

4. Agile methodologies

Several frameworks and methods are used to implement Agile principles, such as:

- **Scrum**: A popular framework that utilizes time-boxed iterations called "sprints," defined roles (Product Owner, Scrum Master, Development Team), and regular meetings (daily stand-ups, sprint reviews, retrospectives).
- **Kanban:** A visual project management system that focuses on workflow visualization, limiting work in progress (WIP), and maximizing efficiency

- **Extreme Programming** (XP): A disciplined approach emphasizing practices like pair programming, test-driven development, and continuous integration to enhance software quality.
- **Lean Software Development**: A methodology focusing on maximizing customer value and eliminating waste in the development process.
- **Scaled Agile Framework (SAFe)**: A framework that integrates Lean-Agile principles for scaling Agile practices across larger organizations and projects.

5. Key practices in agile software development

- **Iterative development**: Breaking down large projects into smaller, manageable chunks and delivering working software in increments.
- **Frequent communication and collaboration**: Encouraging regular interaction among team members, stakeholders, and customers.
- **Continuous feedback and improvement**: Actively seeking and incorporating feedback from users and stakeholders throughout the development cycle.
- **Self-organizing teams**: Empowering development teams to make decisions and manage their work.
- **Transparency**: Keeping the development process and progress visible to all stakeholders.

6. Benefits of the agile approach

- **Increased flexibility and adaptability**: Easily responding to changes in requirements, technology, and market conditions.
- **Faster delivery and quicker time-to-market**: Delivering functional software in shorter cycles.
- **Enhanced quality and customer satisfaction**: Continuous testing and feedback loops lead to better products that meet customer expectations.
- **Improved team collaboration and communication**: Fostering a more engaging and productive work environment.
- **Reduced risk**: Identifying and addressing potential issues early in the process.

7. Challenges in adopting agile

While Agile offers numerous benefits, implementing it successfully can pose challenges, including:

- **Resistance to change**: Overcoming ingrained habits and mindsets from traditional methodologies.
- **Lack of Agile experience**: Teams may need training and coaching to adapt to Agile principles and practices.
- **Inadequate communication**: Maintaining open and transparent communication channels, especially in distributed teams.

- **Insufficient stakeholder involvement**: Ensuring continuous engagement from customers and stakeholders.
- **Cultural misalignment**: Adapting organizational culture to embrace Agile values like collaboration and empowerment.

Agile Process Model

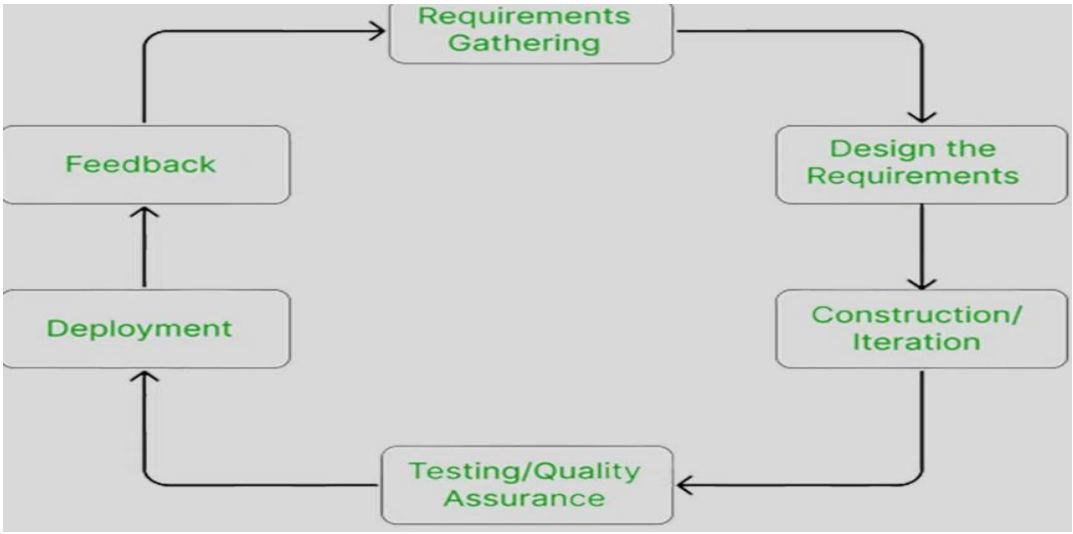
The Agile Process Model was primarily designed to help a project adapt quickly to change requests. So, the main aim of the Agile model is to facilitate quick project completion. To accomplish this task, it's important that agility is required. Agility is achieved by fitting the process to the project and removing activities that may not be essential for a specific project.

Also, anything that is a waste of time and effort is avoided. The Agile Model refers to a group of development processes. These processes share some basic characteristics but do have certain subtle differences among themselves.

Steps in the Agile Model

The Agile Model is a combination of iterative and incremental process models. The phases involve in **Agile (SDLC) Model** are:

1. Requirement Gathering
2. Design the Requirements
3. Construction / Iteration
4. Testing / Quality Assurance
5. Deployment
6. Feedback



1. Requirement Gathering

In this step, the development team must gather the requirements, by interaction with the customer. **development team** should **plan** the **time and effort needed to build the project**. Based on this information you can evaluate technical and economical feasibility.

- Meet with the customer to really understand their needs and what they expect from the software.
- Identify the key requirements and business goals to make sure everyone is on the same page.
- Estimate how much time and effort it will take to develop the software.
- Assess if the project is technically possible and whether it's worth the investment from both a technical and economic standpoint.

2. Design the Requirements

In this step, the development team will use user-flow-diagram or high-level **UML Diagrams** to show the working of the new features and show how they will apply to the existing software. Wireframing and designing user interfaces are done in this phase.

- **Designing the system**: Once the requirements are gathered, the next step is to design the system's overall architecture based on those needs. This helps to verify the software is structured in a way that meets the user's expectations.
- **Creating wireframes**: Next, wireframes for the user interface (UI) are created. These are simple blueprints that show how the software will look and how users will interact with it, ensuring it's user-friendly and easy to navigate.
- **High-level design with UML diagrams**: At this stage, high-level designs using UML (Unified Modeling Language) diagrams are created to visually represent the software's structure and how different parts will work together.
- **Prototyping for feedback**: Prototypes are made to give stakeholders an early look at the software. This helps gather feedback early in the process and allows for adjustments before the full development begins.

3. Construction / Iteration

In this step, development team members start working on their project, which aims to deploy a working product. Each cycle typically consist between **1-4 weeks**, and at the end, the team delivers a working version of the software.

- **Development of Features**: The team works on the features identified during the requirement and design phases.

- **Coding and Implementation**: New functionalities are coded and integrated into the software based on the goals for that specific iteration.
- **Delivering a Working Product**: After each iteration, a usable version of the software is ready.
- **Incremental Improvement**: With every cycle, the software is enhanced, adding more features and refining existing ones.

4. Testing / Quality Assurance

Testing involves Unit Testing, Integration Testing, and System Testing, Which help in the agile development models:

- **Unit Testing:** Unit testing is the process of checking small pieces of code to ensure that the individual parts of a program work properly on their own. Unit testing is used to test individual blocks (units) of code.
- **Integration Testing:** Integration testing is used to identify and resolve any issues that may arise when different units of the software are combined.
- **System Testing:** Goal is to ensure that the software meets the requirements of the users and that it works correctly in all possible scenarios.

5. Deployment

In this step, the development team will deploy the working project to end users.

Once an iteration is finished and fully tested, the software is ready to be released to the end users. In Agile, deployment isn't a one-time event—it's an ongoing process. Updates and improvements are rolled out regularly, making sure the software keeps evolving and getting better with each release.

Deploy the software to a test or live environment so that it can be used by customers or end-users.

Make the software accessible to users, verifying they can start using it as expected.

Verify the deployment goes smoothly and fix any issues that come up quickly

6. Feedback

This is the last step of the **Agile Model.** In this, the team receives feedback about the product and works on correcting bugs based on feedback provided by the customer.

- **Take feedback** from customers, users, and stakeholders after each iteration.
- Understand how well the product meets user needs and identify areas for improvement.
- **Check for bugs** or issues that need fixing.
- **Make adjustments** to the development plan based on feedback to improve the product further.

Benefits of Agile development methodology

The **Advantages of the Agile Model** are as follows:

- **Flexibility and Adaptability**: Agile can quickly adapt to changes, allowing teams to respond to new customer needs and market conditions.
- **Improved Collaboration**: Agile encourages constant communication between developers and stakeholders, ensuring the product meets user expectations.
- **Faster Delivery**: Agile ensures quicker releases, keeping customers engaged and their feedback incorporated early.
- **Enhanced Quality and Customer Satisfaction**: Agile focuses on customer feedback, ensuring the product meets their needs and delivering high-quality results.
- **Iterative Development**: Work is done in small, manageable steps, allowing for regular improvements and quick adjustments.
- **Transparency**: Agile keeps stakeholders informed at every stage, ensuring clarity and alignment.
- **Quality Assurance**: Agile prioritizes quality, ensuring the product meets users' expectations through continuous improvements.
- **Continuous Improvement**: Regular feedback ensures the product keeps improving, preventing last-minute issues and maintaining high quality.

Extreme Programming (XP)

Extreme Programming (XP) is an Agile software development methodology that focuses on delivering high-quality software through **frequent and continuous feedback, collaboration**, and **adaptation.** XP emphasizes a close working relationship between the **development team, the customer, and stakeholders, with an emphasis on rapid, iterative development and deployment.**

Agile approaches are based on some **common principles**, some of which are:

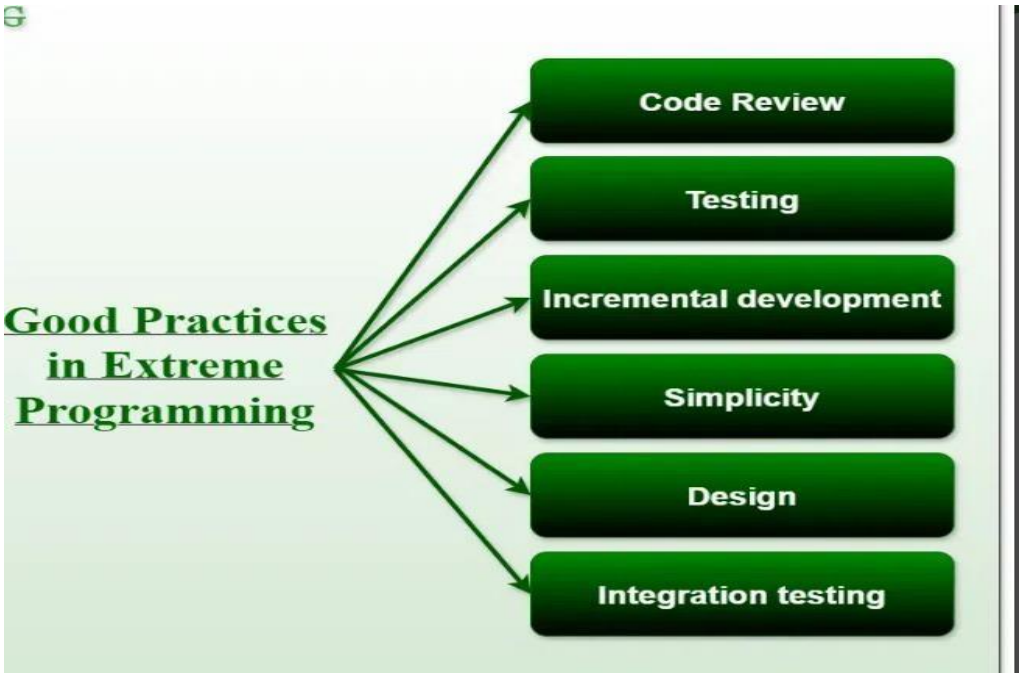
1. Working software is the key measure of progress in a project.
2. For progress in a project, therefore software should be developed and delivered rapidly in small increments.
3. Even late changes in the requirements should be entertained.
4. Face-to-face communication is preferred over documentation.
5. Continuous feedback and involvement of customers are necessary for developing good-quality software.

6. A simple design that involves and improves with time is a better approach than doing an elaborate design up front for handling all possible scenarios.
7. The delivery dates are decided by empowered teams of talented individuals.

Extreme programming is one of the most popular and well-known approaches in the family of agile methods. An XP project starts with user stories which are short descriptions of what scenarios the customers and users would like the system to support. Each story is written on a separate card, so they can be flexibly grouped.

Good Practices in Extreme Programming

Some of the good practices that have been recognized in the extreme programming model and suggested to maximize their use are given below:



- **Code Review:** Code review detects and corrects errors efficiently. It suggests pair programming as coding and reviewing of written code carried out by a pair of programmers who switch their work between them every hour.
- **Testing:** Testing code helps to remove errors and improves its reliability. XP suggests test-driven development (TDD) to continually write and execute test cases. In the TDD approach, test cases are written even before any code is written.
- **Incremental development:** Incremental development is very good because customer feedback is gained and based on this development team comes up with new increments every few days after each iteration.

- **Simplicity:** Simplicity makes it easier to develop good-quality code as well as to test and debug it.
- **Design:** Good quality design is important to develop good quality software. So, everybody should design daily.
- **Integration testing:** Integration Testing helps to identify bugs at the interfaces of different functionalities. Extreme programming suggests that the developers should achieve continuous integration by building and performing integration testing several times a day.

Basic Principles of Extreme programming

XP is based on the frequent iteration through which the developers implement User Stories. User stories are simple and informal statements of the customer about the functionalities needed. A User Story is a conventional description by the user of a feature of the required system. It does not mention finer details such as the different scenarios that can occur. Based on User stories, the project team proposes Metaphors. Metaphors are a common vision of how the system would work. The development team may decide to build a Spike for some features. A Spike is a very simple program that is constructed to explore the suitability of a solution being proposed. It can be considered similar to a prototype.

Some of the **basic activities that are followed during software development by using the XP model** are given below:

- **Coding:** The concept of coding which is used in the XP model is slightly different from traditional coding. Here, the coding activity includes drawing diagrams (modeling) that will be transformed into code, scripting a web-based system, and choosing among several alternative solutions.
- **Testing:** The XP model gives high importance to testing and considers it to be the primary factor in developing fault-free software.
- **Listening:** The developers need to carefully listen to the customers if they have to develop good quality software. Sometimes programmers may not have the depth knowledge of the system to be developed. So, the programmers should understand properly the functionality of the system and they have to listen to the customers.
- **Designing:** Without a proper design, a system implementation becomes too complex, and very difficult to understand the solution, thus making maintenance expensive. A good design results elimination of complex dependencies within a system. So, effective use of suitable design is emphasized.
- **Feedback:** One of the most important aspects of the XP model is to gain feedback to understand the exact customer needs. Frequent contact with the customer makes the development effective.
- **Simplicity:** The main principle of the XP model is to develop a simple system that will work efficiently in the present time, rather than trying to build something that would take time and

may never be used. It focuses on some specific features that are immediately needed, rather than engaging time and effort on speculations of future requirements.

- **Pair Programming:** XP encourages pair programming where two developers work together at the same workstation. This approach helps in knowledge sharing, reduces errors, and improves code quality.
- **Continuous Integration:** In XP, developers integrate their code into a shared repository several times a day. This helps to detect and resolve integration issues early on in the development process.
- **Refactoring:** XP encourages refactoring, which is the process of restructuring existing code to make it more efficient and maintainable. Refactoring helps to keep the codebase clean, organized, and easy to understand.
- **Collective Code Ownership:** In XP, there is no individual ownership of code. Instead, the entire team is responsible for the codebase. This approach ensures that all team members have a sense of ownership and responsibility towards the code.
- **Planning Game:** XP follows a planning game, where the customer and the development team collaborate to prioritize and plan development tasks. This approach helps to ensure that the team is working on the most important features and delivers value to the customer.
- **On-site Customer:** XP requires an on-site customer who works closely with the development team throughout the project. This approach helps to ensure that the customer's needs are understood and met, and also facilitates communication and feedback.

Key Highlights

1. Introduction to Software Process Models
2. Importance of Software Development Life Cycle (SDLC)
3. The Waterfall Model and its phases
4. Advantages and limitations of the Waterfall Model
5. Incremental Process Model and incremental delivery
6. Benefits and applications of the Incremental Model
7. Evolutionary Process Models
8. Prototyping Model
9. Spiral Model
10. Concurrent Development Model
11. Specialized Process Models
12. Component-Based Development (CBD)
13. Formal Methods Model
14. Aspect-Oriented Software Development (AOSD)
15. Agile Software Development Principles
16. Agile Manifesto and Agile Values

17. Agile Process Framework
18. Agile Development Benefits and Challenges
19. Extreme Programming (XP)
20. XP Values (Communication, Simplicity, Feedback, Courage, Respect)
21. XP Practices (Pair Programming, Test-Driven Development, Refactoring, Continuous Integration, Small Releases, Collective Ownership)
22. Comparison of Traditional and Agile Process Models
23. Selection of an Appropriate Software Process Model.

Case Study 1: The Waterfall Model

Title : Library Management System Development

Scenario

A university wants to develop a Library Management System. The requirements are clearly defined before development begins and are not expected to change.

Application

The development team follows the Waterfall Model:

- Requirements are collected from the library staff.
- The system is designed.
- Coding is completed.
- Testing is performed.
- The system is deployed.
- Maintenance is provided after deployment.

Outcome

Since the requirements remain stable, the project is completed successfully within the planned schedule and budget.

Lesson Learned

The Waterfall Model is most suitable when requirements are clear, complete, and unlikely to change.

Case Study 2: Incremental Process Model

Title: Online Banking System

Scenario

A bank plans to launch an online banking application quickly while adding advanced features later.

Application

The software is developed in increments:

- Increment 1: User login and account balance.
- Increment 2: Fund transfer.
- Increment 3: Bill payment.
- Increment 4: Loan management.

Outcome

Customers begin using the system after the first release, while additional features are added over time.

Lesson Learned

The Incremental Model provides early delivery and accommodates changing customer needs.

Case Study 3: Evolutionary Process Model (Prototype)

Title: Hospital Appointment System

Scenario

A hospital is unsure about the exact requirements for an online appointment system.

Application

Developers first create a prototype containing:

- Patient registration
- Doctor search
- Appointment booking

Doctors and patients test the prototype and suggest improvements before the final system is developed.

Outcome: The final software better meets user expectations.

Lesson Learned

The Evolutionary Model is ideal when requirements are unclear or expected to evolve.

Case Study 4: Specialized Process Model

Title: E-Commerce Website Using Reusable Components

Scenario

An e-commerce company wants to reduce development time by reusing existing software components.

Application

Developers integrate reusable modules for:

- Payment gateway
- Shopping cart
- User authentication
- Email notification

Outcome: Development is completed faster with reduced cost and improved reliability.

Lesson Learned

Specialized Process Models, such as Component-Based Development, improve productivity through software reuse.

Case Study 5: Agile Process

Title: Food Delivery Mobile Application

Scenario: A startup develops a food delivery application where customer requirements change frequently.

Application

The team follows Agile practices:

- Short development iterations (sprints)
- Daily team meetings
- Frequent customer feedback
- Continuous improvement

Outcome

New features are delivered regularly, and customer satisfaction improves because feedback is incorporated quickly.

Lesson Learned

Agile is best suited for projects with changing requirements and frequent customer interaction.

Case Study 6: Extreme Programming (XP)

Title: College Student Portal

Scenario

A college wants a student portal that can be updated frequently with new academic features.

Application

The XP team follows these practices:

- Pair Programming for coding quality.
- Test-Driven Development (TDD) before implementation.
- Continuous Integration after every code change.
- Refactoring to improve code quality.
- Small Releases every two weeks.
- Continuous customer involvement.

Outcome

The portal is delivered in small, high-quality releases with fewer defects and faster adaptation to new requirements.

Lesson Learned

Extreme Programming (XP) improves software quality, collaboration, and responsiveness to changing requirements.

Text Books

1. Software Engineering: A Practitioner's Approach, 9th Edition, McGraw-Hill Education.
2. Software Engineering, 10th Edition, Pearson Education.

Reference Books

1. Fundamentals of Software Engineering, PHI Learning.
2. Software Engineering, New Age International Publishers.
3. Software Engineering, Narosa Publishing House.
4. Agile Software Development: Principles, Patterns, and Practices, Pearson Education.
5. Extreme Programming Explained: Embrace Change, Addison-Wesley.

Question Number	Questions	Bloom's Level
Unit – II PROCESS MODELS		
PART – A (2 Marks)		
1	What is the main problem with changing requirements in the Waterfall Model?	L1
2	Name the four quadrants of Boehm's Spiral Model. (L1 - Remember)	L1
3	Write down any two core values of the Agile Manifesto. (L1 - Remember)	L1
4	Explain the term "blocking state" in the Waterfall Model. (L2 - Understand)	L2
5	What does "core product" mean in the Incremental Process Model?	L2
6	What is the main purpose of a "Spike Solution" in Extreme Programming (XP)?	L2
7	Give one real-world example of a project where the Waterfall Model is best.	L3
8	How does an XP team calculate task effort during planning?	L3
9	What is the difference between Throwaway and Evolutionary prototyping regarding code reuse?	L4
10	How do the Spiral Model and the RAD Model handle risk differently?	L4
11	Why is the Formal Methods Model a poor choice for a fast-paced mobile app startup?	L5
12	Draw a simple block diagram of a standard 2-week Extreme Programming iteration cycle.	L6
13	Create a 3-point checklist to help a manager choose between Incremental and Specialized models.	L6
PART – B(10 Marks)		
1	Name all phases of the Waterfall Model and list four types of projects that use it.	L1
2	List the four core values and all twelve principles of the Agile process framework.	L1
3	Explain how the Incremental Process Model works and how successive builds are integrated.	L2
4	Draw a neat diagram of Boehm's Spiral Model and explain why it is called a meta-model.	L2
5	Show how to set up a 60-day RAD model timeline for a university management system.	L3
6	Show how TDD, Pair Programming, and Continuous Integration work together in an XP project.	L3
7	Break down the Component-Based Development model into steps and explain its impact on architecture.	L4
8	Compare the Formal Methods Model with Cleanroom engineering regarding math proof, bugs, and cost.	L4
9	Compare the Waterfall model against Agile philosophy regarding scope creep and late changes.	L5
10	Evaluate how XP practices like Refactoring, Sustainable Pace, and Collective Ownership reduce long-term costs.	L5
11	Design a new hybrid process framework that combines the Spiral Model with Extreme Programming.	L6
12	Build a complete Agile team structure defining all roles, sprint meetings, and velocity metrics.	L6

UNIT - III : SOFTWARE REQUIREMENTS AND SYSTEM MODELS

"A software requirement specifies what the software must do, not how it should do it."

-Roger S. Pressman

Unit Overview

This unit introduces the fundamentals of Requirements Engineering, which focuses on identifying, documenting, validating, and managing software requirements. It covers functional and non-functional requirements, user and system requirements, interface specification, Software Requirements Specification (SRS), feasibility studies, requirements engineering activities, and system modelling techniques such as context, behavioural, and data models using structured methods.

Objectives of the Unit

After studying this unit, students will be able to:

1. Understand the importance of software requirements.
2. Differentiate functional and non-functional requirements.
3. Explain user requirements and system requirements.
4. Understand interface specification and SRS documentation.
5. Evaluate project feasibility before development.
6. Apply requirements elicitation, analysis, validation, and management techniques.
7. Understand context, behavioral, and data models.
8. Learn structured methods for software analysis and design.

Learning Outcomes

At the end of this unit, students will be able to:

1. Identify different types of software requirements.
2. Prepare a Software Requirements Specification (SRS).
3. Conduct feasibility studies for software projects.
4. Apply techniques for gathering and validating requirements.
5. Manage changing software requirements effectively.
6. Develop context, behavioural, and data models.
7. Use structured methods to analyze software systems.

Importance of Studying this Unit

- Provides a strong foundation for successful software development.
- Helps understand customer needs accurately.
- Reduces software development risks and project failures.
- Improves software quality and customer satisfaction.
- Supports effective communication between stakeholders and developers.
- Enables proper documentation and system modeling.
- Facilitates efficient maintenance and future enhancements.

Key Concepts

- Requirements Engineering
- Functional Requirements
- Non-Functional Requirements
- User Requirements
- System Requirements
- Interface Specification
- Software Requirements Specification (SRS)
- Feasibility Study
- Requirements Elicitation
- Requirements Analysis
- Requirements Validation
- Requirements Management
- Context Model
- Behavioral Model
- Data Model
- Structured Methods

Functional and Non Functional Requirements

Requirements analysis is an essential process that enables the success of a system or software project to be assessed. Requirements are generally split into two types: Functional and Non-functional requirements. functional requirements define the specific behaviour or functions of a system. In contrast, non-functional requirements specify how the system performs its tasks, focusing on attributes like performance, security, scalability, and usability.

Functional Requirements

These are the requirements that the end user specifically demands as basic facilities that the system should offer. All these functionalities need to be necessarily incorporated into the system as a part of the contract.

- These are represented or stated in the form of input to be given to the system, the operation performed and the output expected.
- They are the requirements stated by the user which one can see directly in the final product, unlike the non-functional requirements.
- **Testing:** Functional testing (e.g., API testing, integration testing) verifies that the system behaves as expected.

Example: Online Banking System

- Users should be able to log in with their username and password.
- Users should be able to check their account balance.
- Users should receive notifications after making a transaction.

Non-Functional Requirements

These are the quality constraints that the system must satisfy according to the project contract. The priority or extent to which these factors are implemented varies from one project to another. They are also called non-behavioral requirements. They deal with issues like:

- Portability
- Security
- Maintainability
- Reliability
- Scalability
- Performance
- Reusability
- Flexibility

Example: Online Banking System

- The system should respond to user actions in less than 2 seconds.
- All transactions must be encrypted and comply with industry security standards.
- The system should be able to handle 100 million users with minimal downtime.

Testing:

- Non-functional testing (e.g., performance testing, security testing, usability testing) verifies that the system meets these quality standards.

Importance of both types of requirements

Both functional and non-functional requirements are vital for a successful software product. Functional requirements ensure the system performs necessary tasks and meets business and user objectives, while non-functional requirements dictate the system's quality and efficiency, impacting user satisfaction and long-term viability. A system that functions correctly but is slow or difficult to use can lead to user dissatisfaction.

Balancing the two

Balancing these requirements is crucial, as improving one aspect, like security (non-functional), might affect another, like performance (non-functional). Teams need to make informed decisions and manage potential tradeoffs.

In essence, functional requirements define *what* the system does, while non-functional requirements define *how well* it does it. A well-designed system addresses both to create a product that is both correct and provides a positive user experience.

User requirements

User requirements in software engineering describe what the **end-user needs and expects from a software system.** These requirements are typically **expressed in natural language and focus on the user's perspective and desired functionalities**, rather than technical implementation details. They serve as a bridge between the user's vision and the technical development process.

Key aspects of user requirements

- **Focus on the "what" and "why":** User requirements describe **what the system needs to do from the user's perspective,** without **delving into the technical details of how it will be** implemented. They focus on the **user's goals and the value the system provides**.
- **Expressed in user-friendly language:** They are typically **written in natural language**, making them easily understandable by both **technical and non-technical stakeholders**.
- **Foundation for design and development:** User requirements serve as a essential starting **point for designing and developing the software**, ensuring that the **final product aligns with user expectations.**
- **Examples of documentation methods**: User requirements can be documented using techniques like **user stories, use cases, or within a User Requirements Document (URD).**

Importance of user requirements

- Defining user requirements is important for several reasons:
- This approach ensures the software is **user-centric, leading to greater user satisfaction**.
- It **reduces rework and errors by minimizing uncertainty**.
- It **improves communication and collaboration** among **stakeholders.**
- It provides a basis for **testing and validating the system** against **user expectations**.

Example of User Requirements:

Consider a new online food ordering system. User requirements might include:

- **As a customer, I want to be able to browse a menu of available restaurants and their dishes.** This focuses on the user's action of browsing and the content they expect to see.
- **As a customer, I want to be able to add multiple dishes from different restaurants to a single order.** This specifies a desired functionality related to order creation.
- **As a customer, I want to be able to pay for my order using various payment methods, including credit card and mobile payment options.** This outlines the expected payment flexibility.

- **As a restaurant owner, I want to be able to update my menu items and prices in real-time.** This addresses the needs of a different user role within the system.
- **As a delivery driver, I want to be able to view my assigned deliveries and customer addresses on a map.** This highlights the need for location-based features for a specific user.

These examples illustrate how user requirements focus on the "what" (what the user wants to achieve) rather than the "how" (how the system will technically implement it). They are essential for ensuring the developed software meets the needs and expectations of its intended users.

Interface specification

An interface specification in software engineering is a detailed **description of how different software components, modules, or systems interact with each other**. It defines the **rules, protocols, and data formats for communication,** ensuring seamless integration and interoperability.

Key elements of an interface specification:

- **Interface Name:** A unique identifier for the interface.
- **Purpose/Description:** A high-level overview of the interface's function.
- **Inputs:** Description of data or parameters received by the interface, including data types, ranges, and constraints.
- **Outputs:** Description of data or results returned by the interface.
- **Pre-conditions:** Conditions that must be met before the interface can be invoke.
- **Post-conditions:** Conditions that are guaranteed to be true after the interface's execution.
- **Error Handling:** How the interface handles **errors or exceptions.**
- **Protocols/Communication Mechanisms:** The method used for interaction.

Example of an interface specification

Consider a basic user management system with an interface for creating new users. This interface could be defined as follows:

Interface Name: UserManagementService

Description: This interface defines operations related to user management within the system.

Methods:

userCreate :**Description**: Creates a new user in the system.

Parameters: firstName: String, representing the user's first name.

lastName: String, representing the user's last name.

type: String, representing the user's role or type (e.g., "admin", "regular").

email: String, representing the user's unique email address.

password: String, representing the user's password.

Returns: Integer, the unique identifier (ID) of the newly created user.

Exception Handling:

UserAlreadyExists Exception:
Thrown if a user with the provided first Name and last Name or email already exists.

UserNotAuthorized Exception:
Thrown if the attempting user does not have the required permissions to create a new user.

Example Implementation (Conceptual):

A component, such as a UserService class, would then implement this UserManagementService interface. It provides the concrete logic for creating users, handling exceptions, and ensuring data validity, based on the specifications defined in the interface.

Key elements of this example

- **Clear Method Signatures**: Each method clearly defines its name, parameters (with types), and return type.
- **Detailed Descriptions**: Each method's description explains its purpose and expected behavior.
- **Exception Handling**: Specifies the possible errors or exceptional conditions that might occur and how they should be handled.

By clearly defining the interface in this manner, different developers or teams can implement the UserManagementService independently. They know exactly how to interact with it to create new users and what to expect in terms of data and possible error scenarios.

Software Requirements Document (SRD)

A Software Requirements Document (SRD), also known as a Software Requirements Specification (SRS), is a detailed blueprint outlining the intended behavior, features, functionalities, and constraints of a software system. It's a foundational document created early in the software development lifecycle to align all stakeholders on a common understanding of the project's goals, scope, and technical specifications.

Key sections of an SRD

While the structure might vary slightly, a typical SRD includes these essential sections:

1. **Introduction:**

- Purpose: States the document's objectives and the overall goals of the software project.
- Intended Audience: Defines who will use the software (e.g., end-users, internal teams).
- Intended Use: Explains the software's primary function and the problems it will solve.
- **Product Scope:** Delineates the boundaries and limitations of the project, including features and functions that will (and won't) be included.
- **Definitions and Acronyms:** Provides a glossary of terms used in the document to ensure shared understanding.

2. Overall Description:

- **User Needs**: Describes the needs and expectations of the intended users.
- Assumptions and Dependencies: Outlines any assumptions made during requirements gathering and identifies factors the project relies on.

3. System Features and Requirements:

- **Functional Requirements**: Details the specific features and functions the software *must* possess. These define what the software will *do*.
- **Example:** In an e-commerce application, a functional requirement might be: "Users should be able to add products to their shopping carts".
- These can be further detailed using formats like EARS ("When [event], the system shall [response]") or Behavior-Driven Development (BDD) scenarios.

Non-Functional Requirements: Specifies *how* the software should perform, focusing on quality attributes and constraints like performance, security, usability, and reliability.

- Example: "The e-commerce platform must load product pages in under two seconds".

External Interface Requirements: Describes how the software interacts with external components like user interfaces, hardware, other software systems, and communication protocols.

4. Other Requirements (Optional):

Database Requirements: Details the database structure and storage needs.

Legal and Regulatory Requirements: Outlines any legal or industry compliance mandates.

Internationalization and Localization: Specifies requirements for supporting multiple languages and cultural adaptations.

Importance of a clear SRD

Minimizes Misunderstandings: Ensures all stakeholders (developers, clients, users) have a shared understanding of the project's objectives and functionalities, according to Requiment.

Reduces Project Risks: Helps identify potential issues and conflicts early on, reducing the likelihood of costly rework and delays.

Facilitates Development and Testing: Provides a clear roadmap for the development team and a basis for creating effective test cases.

Supports Future Maintenance and Enhancements: Serves as a valuable reference for understanding the software's initial goals and requirements during future updates and maintenance efforts.

Feasibility Studies

A feasibility study in software engineering is a essential step taken in the initial planning and design phases of a proposed software project. It is essentially a systematic and comprehensive analysis designed to assess the viability and potential success of the proposed software solution. This evaluation helps stakeholders make informed decisions before allocating significant time, resources, and budget to the development process.

Types of Feasibility Study

The feasibility study mainly concentrates on below five mentioned areas. Among these Economic Feasibility Study is most important part of the feasibility analysis and Legal Feasibility Study is less considered feasibility analysis.

1. **Technical Feasibility**: In Technical Feasibility current resources both hardware software along with required technology are analyzed/assessed to develop project. This technical feasibility study gives report whether there exists correct required resources and technologies which will be used for project development. Along with this, feasibility study
   also analyzes technical skills and capabilities of technical team, existing technology can be used or not, maintenance and up-gradation is easy or not for chosen technology etc.
2. **Operational Feasibility:** In Operational Feasibility degree of providing service to requirements is analyzed along with how much easy product will be to operate and maintenance after deployment. Along with this other operational scopes are determining usability of product, Determining suggested solution by software development team is acceptable or not etc.
3. **Economic Feasibility:** In Economic Feasibility study cost and benefit of the project is analyzed. Means under this feasibility study a detail analysis is carried out what will be cost of the project for development which includes all required cost for final development like hardware and software resource required, design and development cost and operational cost and so on. After that it is analyzed whether project will be beneficial in terms of finance for organization or not.
4. **Legal Feasibility:** In Legal Feasibility study project is analyzed in legality point of view. This includes analyzing barriers of legal implementation of project, data protection acts or social media

laws, project certificate, license, copyright etc. Overall it can be said that Legal Feasibility Study is study to know if proposed project conform legal and ethical requirements.

5. **Schedule Feasibility:** In Schedule Feasibility Study mainly timelines/deadlines is analyzed for proposed project which includes how much time teams will take to complete final project which has a great impact on the organization as purpose of project may fail if it can't be completed on time.
6. **Cultural and Political Feasibility:** This section assesses how the software project will affect the political environment and organizational culture. This analysis takes into account the organization's culture and how the suggested changes could be received there, as well as any potential political obstacles or internal opposition to the project. It is essential that cultural and political factors be taken into account in order to execute projects successfully.
7. **Market Feasibility:** This refers to evaluating the market's willingness and ability to accept the suggested software system. Analyzing the target market, understanding consumer wants and assessing possible rivals are all part of this study. It assists in identifying whether the project is in line with market expectations and whether there is a feasible market for the good or service being offered.
8. **Resource Feasibility:** This method evaluates if the resources needed to complete the software project successfully are adequate and readily available. Financial, technological and human resources are all taken into account in this study. It guarantees that sufficient hardware, software, trained labor and funding are available to complete the project successfully.

Requirements elicitation and analysis process

Requirements elicitation is a fundamental step in software engineering (SE) that involves the systematic gathering of information from stakeholders to understand and define the needs and expectations of a software system.

Software engineers interact with various stakeholders during requirement elicitation, including clients, end users, and domain experts. The goal is to collect, analyze, and document their inputs, which will serve as the foundation for designing and developing the software, as shown in the illustration below.

There are four stages in the requirements elicitation and analysis process, as shown in the diagram below.

Stages of requirements elicitation and analysis

1. **Requirements discovery:** Interact with stakeholders to uncover system needs and domain requirements, while also reviewing existing documentation. For example, interviewing customers and analyzing sales data to understand e-commerce platform needs.

2. **Classification and organization:** Group requirements by system architecture, identifying subsystems for clarity. For example, grouping car requirements into engine performance, safety features, and interior design categories.
3. **Prioritization and negotiation:** Resolve conflicts and prioritize requirements through stakeholder consensus, even if it requires compromise. For example, balancing marketing's desire for new features with development's need for system optimization in software development.
4. **Requirements specification:** Document requirements formally or informally to provide a clear foundation for system development. For example, documenting smartphone specifications, including screen size, camera details, and battery life for development guidance.

There are multiple techniques followed in eliciting the requirements. Let's explore them one by one.

Requirements elicitation techniques

1. Scenarios

- **Collaborate on scenarios:** Work closely with stakeholders to develop scenarios, incorporating their insights and details.
- **Scenario variety:** Explore different scenario formats, including text, diagrams, and screenshots, to convey requirements effectively.
- **Structured approach:** Consider structured methods like event scenarios or use cases for a more organized approach.
- **Visual aids:** Use visual elements like diagrams to enhance scenario understanding.
- **Detail capture:** Ensure that scenarios capture all the necessary details for a comprehensive requirement understanding.

2. Interviewing

- **Create a clear questionnaire:** Start with a detailed questionnaire.
- **Involve key people:** Include different stakeholder groups.
- **Ask open questions:** Use open-ended queries for free-flowing insights.
- **Flex your methods:** Adjust to their preferences (in-person, groups, video, email).
- **Listen actively:** Dig deeper with follow-up questions to uncover hidden needs.

3. User stories

- **Use conversational text:** Create brief, customer-focused user stories for initial requirements and agile planning.
- **Embrace agile:** Align user stories with agile methods for flexibility.

- **Customer's voice:** Let customers express system needs in their own words.
- **Variety of prototypes:** Prototypes can be working (e.g., software code or simulations) or nonworking (e.g., storyboards and mock-ups).
- **Keep it short:** Aim for two to four sentences on a compact card.
- **Adapt to project:** Adjust the number of user stories based on project size and methodology.

4. Requirement workshops

- **Focused workshops:** Highly structured sessions with selected stakeholders.
- **Clear objectives:** Define, refine, and finalize business requirements.
- **Swift documentation:** Quickly complete and share documentation for review.
- **Instant confirmation:** Get immediate feedback on requirements.
- **Efficient for large groups:** Ideal for efficiently gathering requirements from large groups.

5. Document analysis

- **Review existing materials:** Start by carefully examining the available documents to grasp the business environment and ensure current solutions are on track.
- **Validate implementation:** Use this approach to confirm the effectiveness of existing solutions.
- **Comprehend business needs:** Dive into documents to gain a clear understanding of business requirements.
- **Documents for evaluation:** Focus on business plans, technical documents, problem reports, and any existing requirement documents.

6. Focus groups

- **Focused discussions:** Focus groups facilitate discussions that target specific subjects, differing from individual interviews.
- **Optimal group size:** Typically, a focus group comprises 6 to 12 participants, with the option to form multiple groups for broader participation.
- **Dynamic interaction:** Active discussions foster an environment conducive to idea exchange.

7. Prototyping

1. **Explore new features:** Prototyping helps uncover and explore new system features.
2. **Architectural insights:** Architects use prototypes, like scale drawings and 3-D models, to uncover and validate customer requirements.
3. **Systems engineering benefit:** Systems engineers also employ prototypes to achieve similar goals in requirement discovery.

8. Brainstorming

1. **Idea generation:** Use brainstorming to generate solutions for specific issues involving domain and subject matter experts.
2. **Diverse ideas:** Encourage multiple ideas for a knowledge repository and a range of choices.
3. **Table discussions:** Hold table discussions—typically held as a roundtable discussion, with all participants given equal time to share their ideas.
4. **Key questions:** Brainstorming answers questions like system expectations, risk factors, rules, issue resolutions, and future issue prevention.

9. Ethnography

1. **Observational approach:** Ethnography immerses an analyst in the work environment to grasp operational processes and derive support needs.
2. **In-depth observation:** Analysts keenly observe daily tasks, uncovering implicit system requirements that mirror real-world work practices, not just formal processes.

10. Surveys/questionnaires

1. **Survey/questionnaire approach:** Stakeholders are given a set of questions to quantify their thoughts, with data analyzed to identify key areas of interest.
2. **Effective question crafting:** Craft questions based on high-priority risks, keeping them direct and unambiguous. Notify participants once the survey is ready and remind them to participate.
3. **Broad data collection:** Easily gather data from a large audience.

Requirements validation

Requirements validation is a crucial process in software development that ensures the gathered requirements accurately reflect the real needs and expectations of the stakeholders and will lead to a successful system.

It answers the question, "Are we building the right product?"

Importance of requirements validation

Requirements validation is important for a successful software project because it:

Minimizes errors and rework: Identifying errors or misinterpretations of requirements early avoids costly and time-consuming rework in later development phases.

Ensures stakeholder satisfaction: By confirming that the documented requirements align with the actual needs of the end-users and other stakeholders, the final product is more likely to meet their expectations, leading to higher satisfaction.

Reduces project risks: It helps identify and address potential issues like technical limitations, resource constraints, or conflicting requirements before they impact the project schedule or budget.

Improves communication: It fosters better communication and collaboration between stakeholders and the development team, leading to a shared understanding of project goals and deliverables.

Prevents scope creep: By establishing clear and well-defined requirements, it helps prevent uncontrolled changes and ensures the project stays focused on its objectives.

Key techniques for requirements validation

Several techniques can be used for requirements validation, often in combination for optimal results:

Requirements Reviews:

This involves a systematic examination of the requirements document by a group of experts, stakeholders, and developers to identify errors, inconsistencies, uncertainty, and omissions.

Example: When developing an online banking application, a review of the requirement stating, "Users can transfer funds between their accounts," reveals the need for specifying the types of accounts involved (checking, savings), transaction limits, and security protocols like two-factor authentication.

Prototyping:

This involves creating a working model or simulation of the system based on the requirements, which stakeholders can interact with to provide feedback.

Example: For a new e-commerce website, a clickable prototype demonstrating the user journey from browsing products to adding to the cart and completing the purchase can be shared with potential customers. This allows gathering feedback on user-friendliness, checkout flow, and any missing features.

Test Case Generation: Developing test cases based on the requirements to assess their testability and identify potential issues during development.

Example: For a requirement stating, "The system must process payments securely," test cases would be created to cover various payment scenarios, including successful transactions, invalid card details, insufficient funds, and potential security threats. Difficulty in creating test cases for a particular requirement may indicate ambiguity or incompleteness in that requirement.

Automated Consistency Analysis: Utilizing tools to automatically check for inconsistencies and errors in requirements specifications, especially for complex systems.

Example: A tool might detect a conflict where one requirement states, "Users can access the system 24/7," while another states, "System will be unavailable for maintenance every Sunday from 2 AM to 4 AM." This discrepancy would then be flagged for resolution.

Walkthroughs: A less formal review where stakeholders and team members "walk through" the requirements document, discussing potential issues and concerns.

Example: For a project management tool, a walkthrough of the requirements related to task management might involve a discussion about how tasks are assigned, updated, and tracked, and whether the proposed workflow aligns with the users' actual practices.

Traceability: Ensuring each requirement can be traced back to its source and linked to design, implementation, and test cases.

Example: For a mobile banking app, a requirements traceability matrix could link the requirement "Users can view their transaction history" to specific design documents, code modules, and test cases that verify the accuracy and completeness of the displayed transaction data.

Checklists: Using predefined checklists to methodically assess requirements against quality standards and identify errors.

Example: A checklist for requirements validation could include items such as "Is the requirement clear and unambiguous?", "Is the requirement complete?", "Is the requirement testable?", and "Does the requirement align with business goals?".

By employing these techniques in conjunction and involving relevant stakeholders throughout the process, software development teams can significantly improve the quality of their requirements, minimize risks, and increase the likelihood of delivering a successful and user-satisfying product.

Requirements Management

Requirements management is the structured process of defining, documenting, tracking, prioritizing, and controlling changes to project requirements throughout the entire software development lifecycle. It is crucial for ensuring that the final software product meets the needs and expectations of all stakeholders involved.

Key aspects of requirements management

Requirements Elicitation: This is the process of gathering information from stakeholders (users, customers, business teams, technical experts) to understand their needs and expectations for the system. Techniques include interviews, surveys, workshops, focus groups, and prototypes.

Requirements Analysis and Definition: After elicitation, the gathered information is analyzed, clarified, and transformed into precise, unambiguous requirements. This involves identifying inconsistencies, conflicts, and missing information, and then specifying them clearly and unambiguously in a format like a Software Requirements Specification (SRS) document or user stories.

Requirements Prioritization: Not all requirements have equal importance. Prioritization helps teams focus on the most critical needs first, especially when resources or time are limited. Methods like the MoSCoW method (Must-have, Should-have, Could-have, Won't-have) can be used for prioritization.

Requirements Traceability: This involves establishing and maintaining links between requirements and other project artifacts, including design elements, code modules, and test cases. A Requirements Traceability Matrix (RTM) is often used for this purpose. Traceability helps to ensure that all requirements are being addressed during development and that the final product can be verified and validated against them.

Requirements Change Management: Requirements are rarely static and often evolve throughout the project lifecycle. Change management defines the process for submitting, reviewing, analyzing the impact of, and approving or rejecting change requests, ensuring that changes are implemented in a controlled and systematic manner.

Requirements Validation and Verification: Validation focuses on ensuring that the requirements reflect the true needs of the stakeholders and will lead to a successful system (i.e., building the right product), IBM states requirements validation ensures that product and development goals are met. Verification confirms that the developed software meets the defined requirements correctly (i.e., building the product right). Techniques like reviews, inspections, and testing are used in both processes.

Example

Imagine a company wants to develop a new online grocery ordering and delivery application.

1. Requirements Elicitation

Interviews: Business analysts interview customers to understand their shopping preferences, delivery expectations, and pain points with existing services. They also interview store managers to understand inventory management, order fulfilment processes, and delivery logistics.

Surveys: Customers could be surveyed about their preferred payment methods, delivery windows, and loyalty program features.

Workshops: A workshop with key stakeholders (customers, delivery personnel, store managers, marketing team) could be conducted to brainstorm features and prioritize functionalities.

Observation: Observing how customers currently shop and how store employees manage orders and deliveries can reveal implicit requirements.

2. Requirements analysis and definition

Based on the collected information, the team defines functional requirements such as: "Users can search for groceries by category and keywords" or "The system must allow users to select a preferred delivery time slot."

Non-functional requirements are also defined: "The system must be available 99.9% of the time," "The application must load product pages within 2 seconds," or "The system must secure customer payment information with encryption".

User stories are also written, such as: "As a busy parent, I want to quickly reorder my weekly groceries, so I can save time on shopping".

A Software Requirements Specification (SRS) document is created to formally document all defined requirements.

3. Requirements prioritization

The team prioritizes requirements based on factors like business value (e.g., core features like adding items to a cart are high priority), risk (e.g., payment gateway security is critical), and feasibility.

For instance, "secure payment processing" would be high priority, while "social media sharing of recipes" might be lower priority for the initial release.

4. Requirements traceability

An RTM is created to link each requirement to its design elements (e.g., database schema for product catalog), code modules (e.g., functions for user registration), and test cases (e.g., test cases to verify user login functionality).

This ensures that all parts of the application are designed, built, and tested to meet the specified requirements.

5. Requirements change management

Mid-project, a new government regulation emerges, requiring specific data privacy protocols for handling customer information.

A change request is submitted, documented, and analyzed for its impact on existing requirements, scope, timeline, and budget. StarAgile details the steps involved in a change control process.

A formal change control board reviews the request and decides whether to approve or reject it.

If approved, the relevant requirements are updated, design and code are modified, and new test cases are created to ensure compliance with the new regulation.

6. **Requirements validation and verification**

Validation: Prototypes of the user interface are built and shown to target users to ensure the design meets their expectations and needs. Feedback is collected to refine the user experience.

Verification: After the application is developed, it undergoes thorough testing to verify that each feature and function operates as specified in the requirements document. This includes unit testing, integration testing, and system testing. Acceptance testing by stakeholders confirms that the system meets the initially defined needs.

By diligently managing requirements throughout the software development process, the company developing the online grocery application increases the chances of creating a product that satisfies customer needs, complies with regulations, and is delivered within the projected budget and timeline.

Context model

In software engineering, a context model is a high-level representation of a system's boundaries and its interactions with the external environment. It provides a big-picture view that shows the system, external entities that interact with it, and the data or information that flows between them.

Context models are particularly valuable during the initial phases of a project, such as system analysis and requirement gathering.

Key components of a context model

- **System:** The software being developed, this is typically placed at the center of the diagram.
- **External entities**: These are elements outside the system boundary that interact with it. **Examples** include users, other software systems, databases, and external hardware devices.
- **Data flows**: These are represented by arrows showing the movement of data or information between the system and the external entities. The arrows can be unidirectional or bidirectional.

Purpose and uses

- **Defines scope**: By illustrating the system's boundaries and its interactions, a context model helps define and focus the project's scope, ensuring that the analysis and development concentrate on the relevant components.
- **Clarifies requirements**: It helps in identifying both functional and non-functional requirements by visualizing how the system interacts with its environment and what data it exchanges.
- **Facilitates communication**: Context models are simple and easy for both technical and non-technical stakeholders to understand, making them an excellent communication tool.

- **Manages dependencies and risks**: By mapping out the system's relationships with external entities, the model helps identify potential dependencies and risks. This allows for proactive planning and mitigation.

Example: A hotel reservation system

Imagine you're designing a new online hotel reservation system. A context model would illustrate the following:

- **Central system**: The Hotel Reservation System.
- **External entities:**
  - Customer: Interacts with the system to search for rooms, make reservations, and pay.
  - Hotel staff: Uses the system to manage bookings, check guests in and out, and view reports.
  - Payment gateway: A third-party system that processes customer payments.
  - Hotel database: An external database that stores information on rooms, bookings, and customer details.
- **Data flows:**
  - The "Customer" sends payment and reservation information to the system.
  - The system sends booking confirmations and payment requests to the "Customer" and "Payment Gateway," respectively.
  - The system sends and receives booking information from the "Hotel staff" and "Hotel database."

This type of diagram provides a clear overview of the system's purpose and its place in the larger operational environment.

Context models can be visualized using simple graphical notation, often referred to as context diagrams.

Behavioral models

A behavioral model in software engineering describes how a system responds to events, focusing on the sequence of actions and interactions between its components.

Common behavioral models include State Machine Diagrams, which show system states and transitions, and Activity Diagrams, which illustrate ordered sets of actions.

An example is an online banking system's behavioral model, which uses a Sequence Diagram to show how a user's login attempt triggers events, leading to a verification process that determines access to their account.

- It focuses on the dynamic aspect of a system: its behavior over time and in response to stimuli.
- It helps understand how the system's components interact to achieve functions.
- It's used in requirements elicitation, design, and testing to ensure the system behaves as expected.

Types of Behavioral Models

- **Use Case Diagrams**: Show user goals and how they interact with the system.
- **Sequence Diagrams**: describe interactions between objects over time, showing the order of message exchanges.
- **Activity Diagrams**: Model the flow of activities and decisions within a process or function.
- **State Machine (or State) Diagrams**: Illustrate the different states a system can be in and the events that cause transitions between these states.

Example: Online Banking Login

Imagine a user trying to log in to their online banking account. A behavioral model, specifically a Sequence Diagram, could illustrate this process:

1. **User Action**: The user enters their username and password in the web browser.
2. **System Event**: The browser sends a "Login Request" event to the banking application.
3. **Component Interaction:**
    - The application receives the request and sends a "Verify Credentials" message to the authentication service.
    - The authentication service checks the credentials against its database.
4. **System Response**:
    - If successful, the authentication service sends a "Credentials Valid" message back to the application.
    - The application then sends an "Access Granted" event, leading the user to their account dashboard.
    - If the credentials are invalid, an "Access Denied" message is sent instead.

This model helps visualize the interactions and sequences of events, ensuring that the system correctly handles both successful and unsuccessful login attempts.

Data models

In software engineering, a data model is a blueprint that describes the structure of data, including its types, attributes, and the relationships between them, within a system or application. It is a key part of the design phase, providing a common understanding for business and technical stakeholders about how data will be organized, stored, and used.

Data models serve as a visual roadmap, facilitating communication and helping to identify and prevent errors before code is written.

Key Aspects of Data Models

Purpose:

To define business and application data requirements, create a visual representation of data elements, and provide a structure for database design and implementation.

Components:

Data models typically include:

Entities: The main objects or concepts in the system (e.g., customers, orders, products).

Attributes: The properties or characteristics of each entity (e.g., a customer's name, an order's date).

Relationships: How entities are connected to each other (e.g., a customer places an order).

Data models are often developed in stages:

Conceptual Data Model: A high-level overview of the data that focuses on business requirements and user needs.

Logical Data Model: A more detailed description of data structures, including entities, attributes, and relationships, but without specifying the physical database implementation.

Physical Data Model: The actual database design, specifying table types, data types, constraints, and other technical details for implementation.

Benefits:

Improved Communication: Provides a common language for business analysts, developers, and other stakeholders.

Better Design: Acts as a blueprint for creating efficient and effective databases.

Reduced Errors: Helps uncover flaws and inconsistencies in data requirements and development plans early on.

Enhanced Data Management: Supports better data organization, accessibility, and quality throughout the system.

How Data Models are Used

Requirements Analysis: They start by capturing and visualizing the data requirements of a software application or business process.

Database Design: The data model serves as the foundation for designing the database structure.

Documentation: They create a clear and concise record of the data within a system.

In software engineering, system models are abstract representations used to understand, design, and communicate different views of a system. Context, behavioral, and data models each provide a distinct perspective.

Structured methods

Structured methods in software engineering are systematic, top-down approaches that break down complex systems into manageable parts using specific notations and graphical tools, such as Structured Analysis (SA) and Structured Design (SD), to improve clarity, maintainability, and quality.

These methods focus on logical thinking, functional decomposition, and process-oriented techniques like Data Flow Diagrams (DFDs) to ensure requirements are accurately captured and software is designed to meet those needs effectively.

Key Principles and Concepts

- **Top-Down Decomposition:**

Complex systems are broken into smaller, simpler, and more manageable components, starting from a high-level overview and progressively detailing lower-level functions.

- **Logical vs. Physical Models:**

Structured methods emphasize the logical structure of a system, separating it from physical implementation details (like specific hardware or software) to allow for greater flexibility and ease of maintenance.

- **Modularity and Hierarchy:**

Programs are designed as a hierarchy of functional, interconnected modules, making them easier to understand, develop, and maintain.

- **Structured Programming:**

This involves a disciplined approach to writing code, focusing on clarity and simplicity through the use of control constructs rather than unrestricted jumps (like goto).

Core Techniques and Tools

Structured Analysis (SA):

- **Purpose:** To understand and document system requirements by dividing a complex system into smaller parts.
- **Tools:** Uses graphical tools like Data Flow Diagrams (DFDs) to show how data moves through the system.
- **Outcome:** Produces user-friendly system specifications that clearly define user needs.

Structured Design (SD):

- **Purpose:** To influence the structure of the final program, creating a hierarchical design from a functional decomposition.

- **Tools:** Uses techniques like top-down design and structure charts to visualize the system's components and their relationships.
- **Outcome:** A well-structured and hierarchical design that is easier to implement and maintain.

Data Flow Diagrams (DFDs):

- **Purpose:** To represent the flow of data within a system, showing how data enters, is processed, and exits.
- **Visuals:** Consist of "bubbles" (processes) and arrows (data flow) to illustrate system functions.

Data Dictionary:

A centralized repository for data definitions and structures used in the system,
ensuring consistency in data representation.

Benefits

- **Improved Software Quality:** Leads to more understandable, reliable, and maintainable software.
- **Enhanced Clarity:** Graphical representations and systematic documentation make complex systems easier to grasp.
- **Better Resource Utilization:** Helps design systems that make the best use of available resources.
- **Facilitated Modifications:** Creates flexible systems that can be more easily adapted to future changes.

Case Study

Case Study 1: Online Library Management System

Related Topics:

- Functional and Non-functional Requirements
- User Requirements
- System Requirements

Case Study:

A college library wants to replace its manual book issuing process with an online system. Students should be able to search books, reserve books, borrow and return books online. Librarians should manage book records, members, and overdue fines.

Tasks:

- Identify functional requirements.

- Identify non-functional requirements.
- Write user requirements.
- Write system requirements.

Case Study 2: Hospital Management System

Related Topics:

- Interface Specification
- Software Requirements Specification (SRS)

Case Study:
A hospital plans to develop a management system that allows patients to book appointments online. Doctors can update prescriptions, and administrators manage patient records and billing.

Tasks:

- Design the user interface requirements.
- Identify external interfaces.
- Prepare the major sections of an SRS document.

Case Study 3: College Bus Tracking System

Related Topics: Feasibility Study

Case Study:
A university plans to develop a GPS-based bus tracking application that enables students to view the real-time location of college buses through a mobile app.

Tasks:

- Perform technical feasibility.
- Analyze economic feasibility.
- Analyze operational feasibility.
- Decide whether the project should proceed.

Case Study 4: Online Food Delivery Application

Related Topics:

- Requirements Elicitation and Analysis
- Requirements Validation

Case Study:
A startup wants to build an online food delivery application where customers can order food, restaurants receive orders, and delivery partners track deliveries.

Tasks:

- Identify stakeholders.
- List techniques for gathering requirements.
- Analyze customer requirements.
- Validate the collected requirements before development.

Case Study 5: Employee Payroll Management System

Related Topics: Requirements Management

Case Study:
A company develops a payroll system. During development, the HR department requests additional features such as tax calculation updates, biometric attendance integration, and salary reports.

Tasks:

- Identify requirement changes.
- Explain how requirement changes should be managed.
- Discuss version control and change management.
- Explain the importance of maintaining requirement traceability.

Case Study 6: E-Commerce Shopping Website

Related Topics:

- Context Models
- Behavioral Models
- Data Models
- Structured Methods

Case Study:
An online shopping company develops a website where customers can register, browse products, place orders, make payments, and track deliveries.

Tasks:

- Draw the context model for the system.
- Prepare a behavioral model using a use case or activity diagram.
- Design a data model showing Customers, Products, Orders, and Payments.
- Explain how structured analysis methods (such as Data Flow Diagrams and Entity-Relationship Diagrams) can be used during system analysis.

Text Books

1. Software Engineering: A Practitioner's Approach — *Roger S. Pressman & Bruce R. Maxim, 9th Edition, McGraw-Hill Education.*
2. Software Engineering — *Ian Sommerville, 10th Edition, Pearson Education.*
3. Software Engineering — *K.K. Aggarwal & Yogesh Singh, 3rd Edition, New Age International Publishers.*
4. Fundamentals of Software Engineering — *Rajib Mall, 5th Edition, PHI Learning Pvt. Ltd.*
5. Software Engineering: Principles and Practice — *Hans van Vliet, 3rd Edition, John Wiley & Sons (Wiley India).*

Reference Books

1. Software Requirements — *Karl E. Wiegers & Joy Beatty, 3rd Edition, Microsoft Press.*
2. Mastering the Requirements Process — *Suzanne Robertson & James Robertson, 3rd Edition, Addison-Wesley Professional.*
3. Modern Systems Analysis and Design — *Jeffrey A. Hoffer, Joey F. George & Joseph S. Valacich, 8th Edition, Pearson Education.*
4. Object-Oriented Analysis and Design with Applications — *Grady Booch, 3rd Edition, Addison-Wesley Professional.*
5. Applying UML and Patterns — *Craig Larman, 3rd Edition, Pearson Education (Prentice Hall).*
6. Requirements Engineering: Fundamentals, Principles, and Techniques — *Klaus Pohl, 1st Edition, Springer.*
7. Systems Analysis and Design — *Kenneth E. Kendall & Julie E. Kendall, 10th Edition,Education.*
8. Software Engineering — *Richard Fairley, 2nd Edition, McGraw-Hill Education.*

Question Number	Questions	Bloom's Level
UNIT - III : SOFTWARE REQUIREMENTS AND SYSTEM MODELS		
PART – A (2 Marks)		
1	Define functional requirements.	L1
2	What are non-functional requirements?	L1
3	Differentiate between user requirements and system requirements.	L2
4	What is an interface specification?	L1
5	Define Software Requirements Specification (SRS).	L1
6	List the characteristics of a good SRS.	L1
7	What is a feasibility study?	L1
8	Name the types of feasibility studies.	L1
9	What is requirements elicitation?	L1
10	State any two requirements elicitation techniques.	L1
11	What is requirements validation?	L1
12	Define requirements management.	L1
13	What is a context model?	L1
14	Define a behavioral model.	L1
15	What is a data model?	L1
PART – B(10 Marks)		
1	Explain functional and non-functional requirements with suitable examples.	L2
2	Compare user requirements and system requirements with appropriate examples.	L2
3	Describe the interface specification and explain its role in software development.	L2
4	Explain the Software Requirements Specification (SRS) document and discuss its essential characteristics.	L2
5	Discuss the feasibility study and explain its different types with examples.	L2
6	Explain the requirements elicitation and analysis process. Discuss various elicitation techniques used in software engineering.	L3
7	Describe the requirements validation process and explain different validation techniques.	L3
8	Explain requirements management and discuss the importance of change management and requirement traceability.	L3
9	Illustrate context models, behavioral models, and data models with suitable examples.	L3
10	Explain structured methods used in software engineering. Discuss the importance of Data Flow Diagrams (DFDs) and Entity-Relationship (ER) diagrams in system analysis.	L2
11	Analyze the role of requirements engineering in developing high-quality software systems.	L4
12	Develop an SRS outline for an online library management system by identifying functional, non-functional, user, and system requirements.	L6

UNIT - IV: DESIGN ENGINEERING& ARCHITECTURE, TESTING STRATEGIES

"Design is where quality is fostered in software engineering."

— Roger S. Pressman

Unit Overview

This unit introduces the principles of software design and software testing. It covers the software design process, design quality, design concepts, architectural design, data design, architectural styles and patterns, testing strategies, validation testing, system testing, and debugging techniques to develop reliable and maintainable software systems.

Objectives of the Unit

- Understand the software design process and design quality.
- Learn the fundamental design concepts and design models.
- Study software architecture and architectural design techniques.
- Understand data design and architectural styles and patterns.
- Learn software testing strategies and validation techniques.
- Understand system testing and effective debugging practices.

Learning Outcomes

- After completing this unit, students will be able to:
- Explain the software design process and design quality principles.
- Apply design concepts to create effective software designs.
- Develop architectural and data designs for software systems.
- Identify suitable architectural styles and design patterns.
- Apply software testing strategies for quality assurance.
- Perform validation testing, system testing, and debugging effectively.

Importance of Studying this Unit

- Helps develop high-quality and maintainable software designs.
- Improves software reliability through effective testing.
- Enables the creation of scalable and reusable software architectures.
- Reduces software defects through systematic validation and debugging.
- Enhances problem-solving and software quality assurance skills.
- Provides a strong foundation for software development and maintenance.

Key Concepts

- Design Process
- Design Quality
- Design Concepts
- Design Model
- Software Architecture
- Architectural Design
- Data Design
- Architectural Styles
- Architectural Patterns
- Software Testing Strategy
- Conventional Software Testing
- Validation Testing
- System Testing
- Debugging Techniques
- Software Quality Assurance

Introduction: Design Engineering

Design Engineering is the process of defining the architecture, components, interfaces, and other characteristics of a system or component. Before writing any code, software engineers make crucial decisions about how the software will be structured, how its parts will interact, and how it will meet all specified requirements, both functional and non-functional (like performance and security).

The goal is to create a clear, comprehensive, and understandable plan that guides the development team through the construction phase, ensuring the final software is robust, efficient, and easy to maintain.

Key aspects of design engineering

Architectural design

This is the high-level structure, like drawing the main rooms and foundation of a house. It defines the major components of the system and how they connect and communicate (e.g., separating user interface, business logic, and database parts).

Component-level design

This is like designing individual rooms or specific appliances within the house. It details the internal workings of each major component, specifying classes, functions, and data structures.

User interface (UI) design

Focuses on how users will interact with the software, ensuring it's intuitive and user-friendly (e.g., designing screens, buttons, and navigation).

Database design (if needed)

Involves structuring how data will be stored and managed within the system for efficiency and integrity.

DESIGN PROCESS AND DESIGN QUALITY

The software design process

The design process is a fundamental part of the Software Development Life Cycle (SDLC) that bridges the gap between requirements analysis ("what" the system should do) and implementation ("how" it is built). While the specific steps can vary with different methodologies (e.g., Agile vs. Waterfall), a typical process includes the following stages:

1. **Requirements analysis:** Before any design work begins, the team gathers and documents the system's functional and non-functional requirements. This ensures the design addresses user needs, business goals, and potential constraints.

Example: For a new e-commerce application, the requirements would specify what products can be sold (functional), but also that the system must handle 10,000 concurrent users without slowing down (non-functional).

2. **Architectural design:** This is the high-level blueprint that defines the system's major components, their relationships, and the underlying technology stack.

Example: An architect might decide to use micro services architecture to decompose the e-commerce application into smaller, independent services for user authentication, product catalog, and payment processing.

3. **Detailed design:** Following architectural decisions, designers create detailed specifications for each component and module, including data structures, API contracts, and user interface (UI) layouts.

Example: The detailed design for the "payment processing" microservice would specify the REST API endpoints, the database schema for transactions, and the specific third-party payment gateway integration.

4. **UI/UX design:** This step focuses specifically on the user interface and user experience. It involves creating wireframes, mockups, and prototypes to map out how users will interact with the system.

Example: Designers create wireframes for the e-commerce checkout flow, ensuring it is intuitive and requires minimal clicks to complete a purchase.

5. **Design review:** Stakeholders, including developers, designers, and business owners, review the design documents and prototypes. This feedback loop helps identify potential issues early and ensures the design meets everyone's needs.

6. **Prototyping:** For complex or high-risk features, a prototype may be developed to test the design before full-scale implementation. This is especially common in agile environments.

Design Quality

The software architecture and design must address the many qualities the final system is expected to possess. These qualities apply to the system as a whole, transcending specific functions and often constraining design choices.

High-quality design is crucial for creating software that is not only functional but also easy to maintain, adapt, and scale. Key quality attributes include:

Maintainability: The ease with which the software can be understood, repaired, and enhanced.

Good quality example: A modular e-commerce application where a single component, like the "product catalog," can be updated without affecting the "payment processing" component.

- **Usability:** How intuitive and user-friendly the application is for its intended users.
    - **Good quality example:** A flight booking app that is simple to use and has a clear, linear booking process.
- **Reliability:** The probability of the software performing its intended function without failure.
    - **Good quality example:** A banking application with a robust transaction system that always processes payments correctly and can recover gracefully from network interruptions.
- **Scalability:** The ability of the system to handle an increased load or workload without a decrease in performance.
    - **Good quality example:** A cloud-native microservices architecture that can automatically spin up new instances of a service when traffic increases, such as during a flash sale.

- **Extensibility:** The ease with which new functionalities can be added to the software.
    - **Good quality example:** A social media platform designed with a plug-in architecture, allowing developers to easily add a new photo filter feature without altering the core codebase.
- **Security:** How well the application protects information and withstands malicious attacks.
    - **Good quality example:** A software system built with a zero-trust architecture, where no component is trusted by default and all interactions are authenticated and authorized.

Design Concepts

The **software design concept** simply means the idea or principle behind the design. It describes how you plan to solve the problem of designing software, and the logic, or thinking behind how you will design software.

It allows the software engineer to create the model of the system software or product that is to be developed or built. The software design concept provides a supporting and essential structure or model for developing the right software. There are many concepts of software design and some of them are given below:

Points to be Considered While Designing Software

1. **Abstraction** (**Hide Irrelevant data**)**:** Abstraction simply means to hide the details to reduce complexity and increase efficiency or quality. Different levels of Abstraction are necessary and must be applied at each stage of the design process. so that any error that is present can be removed to increase the efficiency of the software solution and to refine the software solution.

2. **Modularity (subdivide the system):** Modularity simply means dividing the system or project into smaller parts to reduce the complexity of the system or project. In the same way, modularity in design means subdividing a system into smaller parts so that these parts can be created independently and then use these parts in different systems to perform different functions. It is necessary to divide the software into components known as modules .

3. **Architecture (design a structure of something):** Architecture simply means a technique to design a structure of something. Architecture in designing software is a concept that focuses on various elements and the data of the structure. These components interact with each other and use the data of the structure in architecture.
4. **Refinement (removes impurities):** Refinement simply means to refine something to remove any impurities if present and increase the quality. The refinement concept of software design is a process of developing or presenting the software or system in a detailed manner which means elaborating a system or software. Refinement is very necessary to find out any error if present and then to reduce it.
5. **Pattern (a Repeated form):** A pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern. The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.
6. **Information Hiding (Hide the Information):** Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party. In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and can't be accessed by any other modules.
7. **Refactoring (Reconstruct something):** Refactoring simply means reconstructing something in such a way that it does not affect the behavior of any other features. Refactoring in software design means reconstructing the design to reduce complexity and simplify it without impacting the behavior or its functions.

Different levels of Software Design

There are three different levels of software design. They are:

1. **Architectural Design:** The architecture of a system can be viewed as the overall structure of the system and the way in which structure provides conceptual integrity of the system. The architectural design identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of the proposed solution domain.

2. **Preliminary or high-level design:** Here the problem is decomposed into a set of modules, the control relationship among various modules identified, and also the interfaces among various modules are identified. The outcome of this stage is called the program architecture. Design representation techniques used in this stage are structure chart and UML.
3. **Detailed design:** Once the high-level design is complete, a detailed design is undertaken. In detailed design, each module is examined carefully to design the data structure and algorithms. The stage outcome is documented in the form of a module specification document.

The Design model

Design models in software engineering serve as blueprints that translate software requirements into a detailed plan for implementation. They provide various perspectives on the system's structure, behavior, and data.

Key design model concepts include:

1. Architectural Design Model:

This model defines the **overall structure of the system, breaking it down into major components, their responsibilities, and how they interact**.

Example: For an e-commerce application, an architectural design might depict a layered architecture with presentation, application, and data layers, showing how user requests flow from the web browser through these layers to the database and back.

2. **Interface Design Model**:

This model specifies how different components of the system, and the system itself, interact with external entities (users, other systems).

Example: In a banking application, the interface design for a user login module would detail the input fields (username, password), validation rules, and the expected output (successful login, error message).

3. **Component-Level Design Model**:

This model focuses on the internal design of individual components, detailing their data structures, algorithms, and how they perform their specific functions.

Example: For the "send message" component in an instant messaging system, the component-level design would specify the data structure for a message object, the algorithm for encrypting the message, and how it interacts with the network transmission module.

4. Data Design Model:

This model describes the data structures required by the software, including how data is stored, organized, and accessed.

Example: In a social media application, the data design might include entity-relationship diagrams (ERDs) showing tables for users, posts, comments, and their relationships, along with data types and constraints for each attribute.

5. **Deployment-Level Design Model**:

This model illustrates how the software components are mapped to physical hardware and network infrastructure.

Example: For a cloud-based application, the deployment design would show how different services (e.g., web server, database server, microservices) are deployed on virtual machines or containers, and how they communicate across a network.

These design models often utilize standardized notations like the Unified Modeling Language (UML) to represent their concepts diagrammatically, facilitating clear communication and understanding among development teams.

Creating an architectural design: Software Architecture

The software needs an architectural design to represent the design of the software. IEEE defines architectural design as "the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system." The software that is built for computer-based systems can exhibit one of these many architectural styles.

Software Architecture defines **fundamental organization** of a system and more simply defines a structured solution. It determines how the various components of a software system are assembled, how they relate to one another, and how they communicate. Essentially, it serves as a **blueprint** for the application and a foundation for the **development team** to build upon.

Software architecture defines a list of things which results in making many things easier in the software development process.

- **System structure**: The organization and arrangement of components.
- **System behavior**: The expected functionality and performance.
- **Component relationships**: How different parts of the system interact.
- **Communication structure**: The way components communicate with each other.
- **Stakeholder balance**: Meeting the needs and expectations of all stakeholders.
- **Team structure**: How the development team is organized and coordinated.
- **Early design decisions**: Making important choices early on to guide development.

Key Characteristics of Software Architecture

Software architecture is a multifaceted concept, and architects often categorize its characteristics based on various factors such as operation, requirements, and structure.

Operational Architecture Characteristics

- **Availability**: The system should be accessible when needed.
- **Performance**: The system should meet performance goals such as speed and responsiveness.
- **Reliability**: The system should work consistently without failure.

- **Fault tolerance**: The system should gracefully handle errors and failures.
- **Scalability**: The system should be able to handle increasing loads without performance degradation.

Structural Architecture Characteristics

- **Configurability**: The ability to configure the system according to needs.
- **Extensibility**: The system should be easily extendable to add new features.
- **Supportability**: The ease with which the system can be maintained and supported.
- **Portability**: The system should work across different environments.
- **Maintainability**: The system should be easy to update and fix over time.

Cross-Cutting Architecture Characteristics

- **Accessibility**: Ensuring the system is usable by a wide range of people, including those with disabilities.
- **Security**: Protecting the system from unauthorized access and data breaches.
- **Usability**: Ensuring the system is easy to use and intuitive for users.
- **Privacy**: Protecting users' sensitive information.
- **Feasibility**: The system should be realistic to develop within the constraints.

Software Architecture Important

Software architecture comes under **design phase** of software development life cycle. It serves as one of the first steps in the software development process. Without software architecture proceeding to software development is like building a house without designing architecture of house. So software architecture is one of important part of software application development. In **technical** and **developmental** aspects point of view below are reasons software architecture are important.

- **Optimizes quality attributes**: Architects select quality attributes to focus on, such as performance and scalability.
- **Facilitates early prototyping**: Architecture allows for early prototypes to be built, offering insight into system behavior.
- **Component-based development**: Systems are often built using components, which makes them easier to develop, test, and maintain.
- **Adapts to changes**: Architecture helps manage and integrate changes smoothly throughout the development process.

Besides all these **software architecture** is also important for many other factors like quality of software, reliability of software, maintainability of software, Supportability of software and performance of software and so on.

Advantages of Software Architecture

Good software architecture provides several advantages:

- **Solid foundation**: It lays the groundwork for a successful project, guiding development.
- **Improved performance**: A well-designed architecture can improve system efficiency.
- **Reduced costs**: Efficient development practices reduce costs over time.
- **Scalability and flexibility**: The system can adapt to future changes or demands.

Disadvantages of Software Architecture

While software architecture is essential, it does come with some challenges:

- **Tooling and standardization**: Obtaining the right tools and maintaining consistent standards can sometimes be a challenge.
- **Uncertain predictions**: It's not always possible to predict the success of a project based solely on its architecture.

Data Design

Data design in software engineering focuses on structuring and organizing data within a system to ensure efficiency, integrity, and maintainability.

Key concepts and their examples include:

Data Modeling:

The process of creating a visual representation of the data within a system, showing entities, attributes, and their relationships.

Example: In an e-commerce system, a conceptual data model might show entities like "Customer," "Product," and "Order," with attributes like

"customer ID," "product name," and "order date," and relationships like "a customer places an order" and "an order contains products."

Data Structures:

The specific ways data is organized and stored to facilitate efficient access and manipulation. **Example:** Using a LinkedList to store a sequence of items where frequent insertions and deletions are expected, or a HashMap for fast lookups based on a key (e.g., storing user profiles by user ID).

Data Hiding (Encapsulation):

The principle of restricting direct access to an object's internal data and exposing only necessary interfaces.

Example: A BankAccount class might have private attributes
for balance and accountNumber, with public methods like deposit(), withdraw(),
and getBalance() to interact with them, preventing direct manipulation of the balance.

Database Design:

The process of structuring a database to store and retrieve data efficiently, considering different types of databases.

Relational Database Example: Designing tables in a SQL database for a library system, with Books table (ISBN, Title, Author), Members table (MemberID, Name,

Address), and a Loans table linking them (LoanID, BookISBN, MemberID, LoanDate, ReturnDate).

NoSQL Database Example: Using a document database like MongoDB to store flexible Product documents, each containing product details, reviews, and related images, without a rigid schema.

Data Integrity:

Ensuring the accuracy, consistency, and reliability of data throughout its lifecycle.

Example: Implementing validation rules in a web form to ensure users enter a valid email address format or setting a NOT NULL constraint on a database column to prevent empty values.

Data Flow Diagrams (DFDs):

Graphical representations that illustrate how data moves through a system, from input to output.

Example**:** A DFD for an online order processing system showing data flowing from "Customer" (input) to "Order Processing" (process), then to "Inventory Management" and "Payment Gateway," and finally to "Shipping" (output).

Architectural Styles and Patterns:

Software architecture styles and patterns provide frameworks for organizing systems, with styles offering broad concepts and patterns detailing specific solutions to common problems. Key examples include **Layered architecture** for organizing code into tiers, **Microservices** for building applications as small, independent services, **Event-Driven Architecture (EDA)** for real-time systems, and the **Client-Server** model for requests and responses.

Architectural Styles

These are high-level ways to structure an application, defining modules and their relationships.

Layered (N-Tier) Architecture: Organizes code into horizontal layers, such as presentation, business logic, and data access.

Example: A typical e-commerce application where the front-end (presentation layer) interacts with the back-end (business logic), which in turn communicates with a database (data access layer).

Client-Server:A system where clients request services from a central server.

Example: Web browsers (clients) requesting web pages from web servers.

Service-Oriented Architecture (SOA): Structures an application as a collection of loosely coupled, reusable services that communicate via well-defined interfaces.

Example: A bank's system where separate services handle account management, funds transfers, and customer information, all interacting with each other.

Monolithic Architecture: An application built as a single, large, indivisible unit, with all its components bundled together.

Example: A simple web application deployed as a single executable or web archive (WAR file).

Microservices Architecture: A modern evolution of SOA, decomposing an application into small, independent services that are deployed and managed separately.

Example: A large e-commerce platform with separate microservices for user authentication, product catalog, order processing, and payment.

Architectural Patterns

These are specific, repeatable solutions to common problems within a given architectural style.

Model-View-Controller (MVC): Separates an application into three interconnected components: Model (data and business logic), View (user interface), and Controller (handles user input).

Example: A mobile app or web application where the UI (View) is updated based on data from the backend (Model), all coordinated by user actions (Controller).

Event-Driven Architecture (EDA):A design pattern where components communicate asynchronously by producing and reacting to events.

Example: An e-commerce system that publishes an "Order Placed" event, which then triggers other services to handle inventory, shipping, and notifications.

Pipe-Filter Pattern: Data flows through a series of processing steps (filters) connected by pipes, transforming the data at each stage.

Example: A data processing pipeline that reads data, validates it, transforms it into a specific format, and then writes it to a database.

Hexagonal Architecture (Ports and Adapters): Decouples the core application logic from external concerns (like user interfaces or databases) through well-defined ports and adapters.

Example: An application that can easily switch between different database types or different user interfaces by swapping out their respective adapters without changing the core business logic.

Architectural design

The software needs an architectural design to represent the design of the software. IEEE defines architectural design as "the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system." The software that is built for computer-based systems can exhibit one of these many architectural styles.

Taxonomy of Architectural Styles

1] Data centered architectures:

- A data store will reside at the center of this architecture and is accessed frequently by the other components that update, add, delete, or modify the data present within the store.
- The figure illustrates a typical data-centered style. The client software accesses a central repository. Variations of this approach are used transform the repository

into a blackboard when data related to the client or data of interest for the client change the notifications to client software.

- This data-centered architecture will promote integrability. This means that the existing components can be changed and new client components can be added to the architecture without the permission or concern of other clients.
- Data can be passed among clients using the blackboard mechanism.

Advantages of Data centered architecture:

- Repository of data is independent of clients
- Client work independent of each other
- It may be simple to add additional clients.
- Modification can be very easy

Data flow architectures

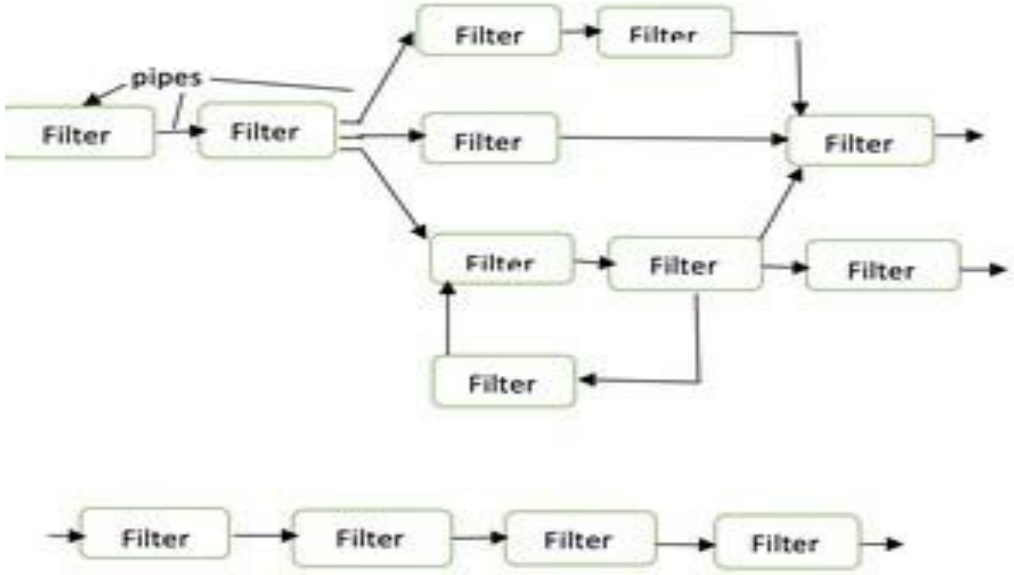
- This kind of architecture is used when input data is transformed into output data through a series of computational manipulative components.
- The figure represents pipe-and-filter architecture since it uses both pipe and filter and it has a set of components called filters connected by lines.
- Pipes are used to transmitting data from one component to the next.
- Each filter will work independently and is designed to take data input of a certain form and produces data output to the next filter of a specified form. The filters don't require any knowledge of the working of neighboring filters.
- If the data flow degenerates into a single line of transforms, then it is termed as batch sequential. This structure accepts the batch of data and then applies a series of sequential components to transform it.

Advantages of Data Flow architecture:

- It encourages upkeep, repurposing, and modification.
- With this design, concurrent execution is supported.

Disadvantage of Data Flow architecture:

- It frequently degenerates to batch sequential system
- Data flow architecture does not allow applications that require greater user engagement.
- It is not easy to coordinate two different but related streams .

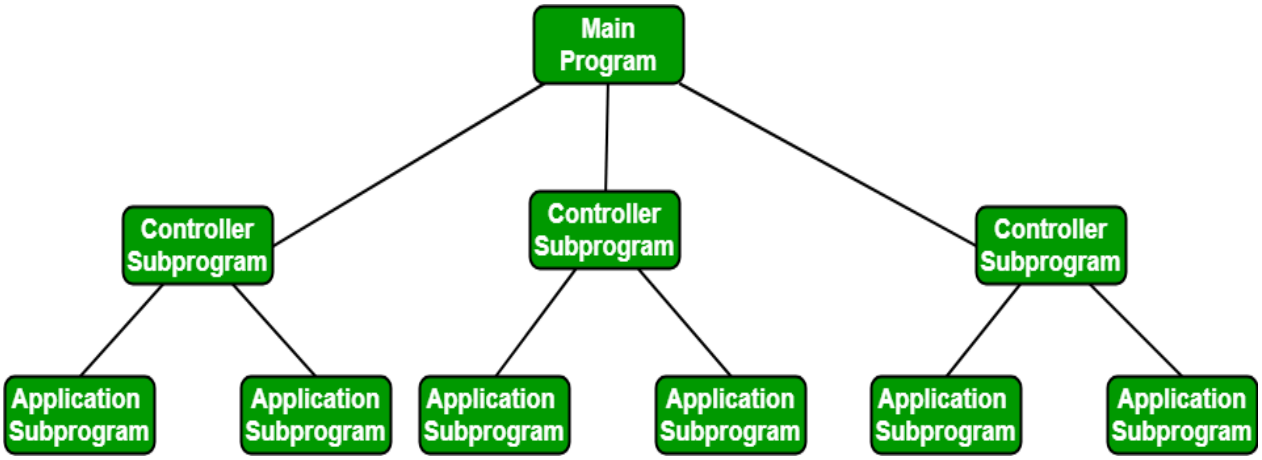


Data Flow architecture

3] Call and Return architectures

It is used to create a program that is easy to scale and modify. Many sub-styles exist within this category. Two of them are explained below.

- **Remote procedure call architecture:** This components is used to present in a main program or sub program architecture distributed among multiple computers on a network.
- **Main program or Subprogram architectures:** The main program structure decomposes into number of subprograms or function into a control hierarchy. Main program contains number of subprograms that can invoke other components.



4] Object Oriented architecture

The components of a system encapsulate data and the operations that must be applied to manipulate the data. The coordination and communication between the components are established via the message passing.

Characteristics of Object Oriented architecture:

- Object protect the system's integrity.
- An object is unaware of the depiction of other items.

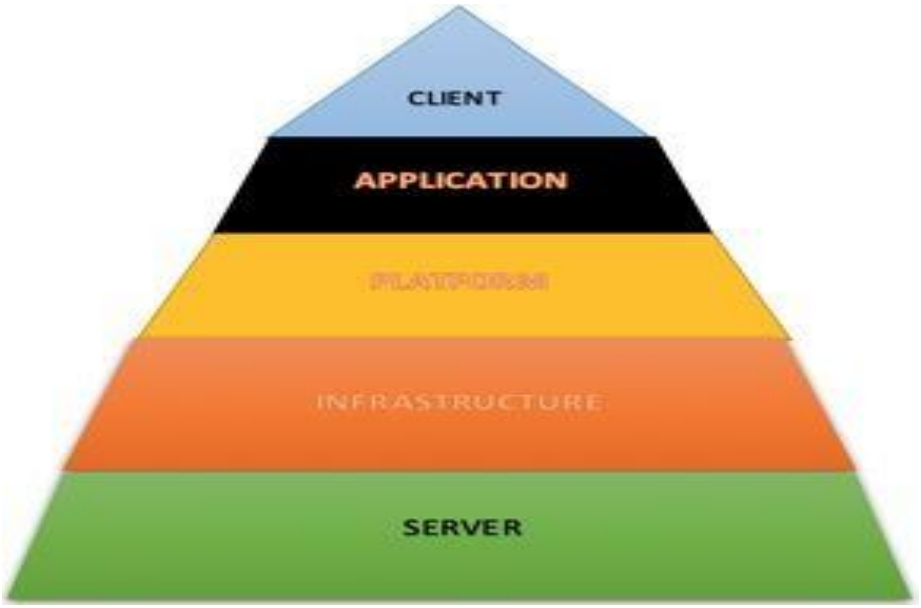
Advantage of Object Oriented architecture:

- It enables the designer to separate a challenge into a collection of autonomous objects.
- Other objects are aware of the implementation details of the object, allowing changes to be made without having an impact on other objects.

5] Layered architecture

- A number of different layers are defined with each layer performing a well-defined set of operations. Each layer will do some operations that becomes closer to machine instruction set progressively.
- At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing(communication and coordination with OS)

- Intermediate layers to utility services and application software functions.
- One common example of this architectural style is OSI-ISO (Open Systems Interconnection-International Organisation for Standardisation) communication system.



Layered architecture

Strategic approaches to software testing

Strategic approaches to software testing include **risk-based testing**, **requirements-based testing**, **analytical strategy**, **model-based testing**, and **methodical approaches**. A risk-based approach, for example, prioritizes testing for the most critical or high-risk functions first, such as user authentication, while a requirements-based strategy ensures every documented requirement is met through specific test cases.

Examples of strategic approaches

1. Risk-based testing

- **Strategy:** This is an analytical approach that focuses testing efforts on areas that pose the highest risk to the project or users.
- **Example:** In an e-commerce application, the team would prioritize testing the payment processing, user login, and credit card information handling, as failures

in these areas have the most severe consequences. Less critical features like "wish list" functionality would receive less attention initially.

2. Requirements-based testing

- **Strategy:** This strategy ensures that the software meets all specified requirements by designing test cases directly from the requirements documentation.
- **Example:** If a requirement states, "The password must be at least 8 characters long and include one uppercase letter and one number," the test cases would be specifically designed to check this requirement. Tests would be created to verify that passwords that meet the criteria are accepted, and those that don't are rejected with an appropriate error message.

3. Model-based testing

- **Strategy:** A model is created to represent the software's structure or behavior, and test cases are automatically generated from this model.
- **Example:** For a state-machine-based feature like a shopping cart, a model could represent states like "empty," "items added," and "checked out." Test cases would then be automatically generated to navigate through these states, ensuring the system behaves correctly at each transition (e.g., moving from "items added" to "checked out" after payment).

4. Methodical testing

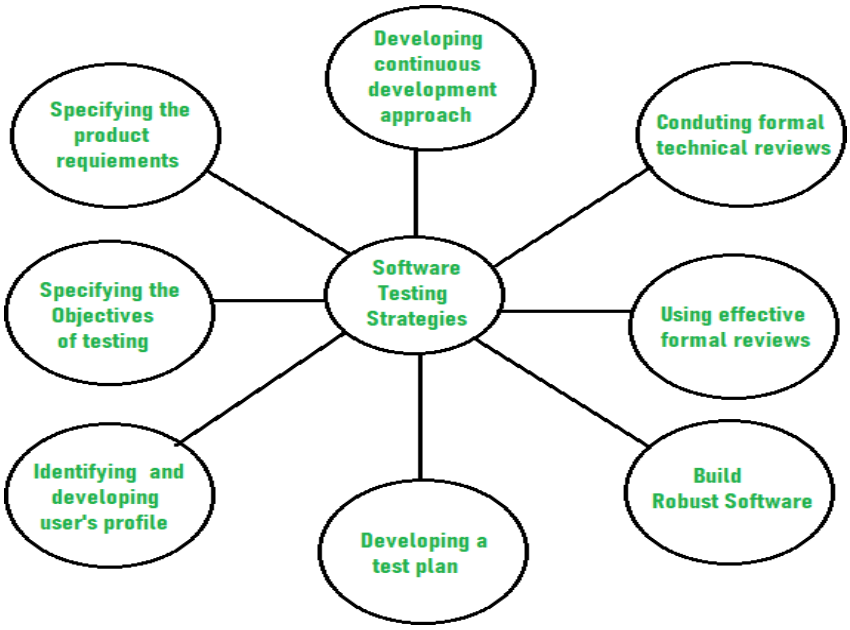
- **Strategy:** This is a combination of structured methodologies and systematic procedures to ensure comprehensive testing.
- **Example:** A project might use a methodical approach that includes:
    - **Unit testing** for individual components.
    - **Integration testing** to ensure components work together.
    - **System testing** on the complete system.
    - **Acceptance testing** with the end-user to validate requirements from a user perspective.

5. Dynamic and heuristic approaches

- **Strategy:** Dynamic testing involves observing and evaluating the software's performance and behavior while it's running. Heuristic approaches involve experience and intuition to find defects.
- **Example:** A dynamic approach is used in **performance testing** to assess how a web application performs under a heavy user load. A heuristic approach is used in **exploratory testing**, where testers simultaneously learn about the system, design tests, and execute them based on their intuition to find bugs that might not have been covered by scripted tests.

Software testing Strategies

The main **objective** of software testing is to **design the tests** in such a way that it **systematically finds different types of errors without taking much time** and effort so that less time is required for the development of the software. **The overall strategy for testing software includes:**



Software testing Strategies

Before testing starts, it's necessary to identify and specify the requirements of the product in a quantifiable manner. Different characteristics quality of the software is there such as maintainability that means the **ability to update and**

modify, the **probability** that means **to find and estimate any risk**, and **usability** that means how it can **easily be used by the customers or end-users**. All these characteristic qualities should be specified in a particular order to obtain clear test results without any error.

1. **Specifying the objectives of testing in a clear and detailed manner.** Several objectives of testing are there such as **effectiveness** that means how effectively the software can achieve the target, any failure that means **inability to fulfill the requirements and perform functions,** and the cost of defects or errors that mean the cost required to fix the error. All these objectives should be clearly mentioned in the test plan.

2. **For the software, identifying the user's category and developing a profile for each user.** Use cases describe the interactions and communication among different classes of users and the system to achieve the target. So as to identify the actual requirement of the users and then testing the actual use of the product.

3. **Developing a test plan to give value and focus on rapid-cycle testing.** Rapid Cycle Testing is a type of test that improves quality identifying and measuring the any changes that need to be required for improving the process of software. Therefore, **a test plan** is an important and **effective document that helps the tester to perform rapid cycle testing.**

4. **Robust software is developed that is designed to test itself.** The software should be capable of **detecting or identifying different classes of errors.** Moreover, software design should allow automated and regression testing which tests the software to find out if there is any adverse or side effect on the features of software due to any change in code or program.

5. **Before testing, using effective formal reviews as a filter.** Formal technical reviews is technique **to identify the errors** that are not discovered yet. The effective technical reviews conducted before testing reduces a significant amount of testing efforts and time duration required for testing software so that the overall development time of software is reduced.

Test strategies for conventional software

Conventional testing is defined as traditional testing where the main aim is to check whether all the requirements stated by the user are achieved.

- The difference between conventional testing and other testing approach is that it concentrates on checking all the requirements given by the user rather than following a software development life cycle.
- Conventional testing mainly focuses on functional testing.
- This testing is being performed by a dedicated team of software testers.

Test strategies for conventional software in software engineering involve a systematic approach to ensure the quality and functionality of the software. These strategies often follow a hierarchical model, moving from testing individual components to testing the entire integrated system.

1. Unit Testing:

This strategy focuses on testing individual units or components of the software in isolation. The goal is to verify that each unit functions correctly according to its specifications.

Example: Testing a specific function in a calculator application to ensure it correctly performs addition, subtraction, multiplication, or division for various inputs.

2. Integration Testing:

After unit testing, this strategy focuses on testing the interactions and interfaces between different integrated units or modules. The aim is to uncover defects arising from the combination of components.

Example: In an e-commerce application, testing the integration between the user authentication module and the shopping cart module to ensure a logged-in user can add items to their cart seamlessly.

3. System Testing:

This strategy evaluates the complete and integrated software system to verify that it meets the specified requirements. It often includes functional and non-functional testing aspects.

Example: Testing an entire banking system to ensure all features (account management, transactions, security) work as expected and that the system performs well under expected load.

4. Validation Testing (Acceptance Testing):

This final phase of testing ensures that the software meets the user's or customer's requirements and expectations. It is often conducted by end-users or stakeholders.

Example: A client using a newly developed project management tool to verify that it addresses their specific workflow needs and provides the required reporting functionalities.

5. Regression Testing:

This strategy involves re-executing previously passed test cases after changes or modifications have been made to the software. Its purpose is to ensure that the changes have not introduced new defects or negatively impacted existing functionalities.

Example: After adding a new feature to a social media application, re-running tests for core functionalities like posting, liking, and commenting to ensure they still work correctly.

6. Performance Testing:

This strategy assesses the software's performance characteristics, such as speed, scalability, stability, and resource usage under various load conditions.

Example: Testing a website's response time and stability when accessed by a large number of concurrent users to identify potential bottlenecks.

Validation testing

Validation testing in software engineering focuses on confirming that the software meets the user's requirements and expectations, essentially ensuring "are we building the right system?". This differs from verification, which confirms "are we building the system right?".

Here are key types of validation testing with examples:

User Acceptance Testing (UAT):

This is performed by end-users or client stakeholders to ensure the system meets their business needs and is fit for purpose in a real-world scenario.

- **Example:** A client tests a new online banking application, performing transactions, checking balances, and paying bills to ensure it functions as expected for their daily financial activities.

Functional Testing:

This verifies that each function of the software operates according to the specified requirements. It's often black-box testing, focusing on inputs and outputs without considering internal code structure.

- **Example:** Testing a login feature to ensure that valid credentials grant access and invalid credentials result in an error message.

System Testing:

This tests the entire integrated system to ensure it meets the specified requirements and functions correctly as a whole. It includes testing interactions between different modules and external systems.

- **Example:** Testing an e-commerce website end-to-end, from adding items to the cart, through checkout, to order confirmation and inventory updates.

Alpha Testing:

This is an internal validation testing phase, typically conducted by a small group of internal testers (often developers and QA) before external release.

- **Example:** A software development team tests a new feature internally, identifying and reporting bugs before it's released to a wider beta testing group.

Beta Testing:

This involves releasing a pre-release version of the software to a select group of external users (beta testers) in a real-world environment to gather feedback and identify issues.

- **Example:** A company releases a beta version of a new mobile app to a group of early adopters, who use it and provide feedback on usability, performance, and bugs.

Operational Acceptance Testing (OAT):

This ensures that the system is ready for operational use, covering aspects like backup and recovery, security, and performance under production conditions.

- **Example:** Testing the backup and restore procedures for a critical database system to ensure data can be recovered effectively in case of a failure.

Regression Testing:

This is performed to ensure that new changes or additions to the software have not adversely affected existing functionalities or introduced new defects.

Example: After adding a new payment gateway to an e-commerce platform, regression tests are run to confirm that existing payment methods still function correctly and no new bugs have been introduced in other areas of the checkout process.

System testing

System testing in software engineering involves evaluating a fully integrated and complete software system to ensure it meets the specified requirements. It is a crucial phase that comes after unit and integration testing, focusing on the system's overall functionality, performance, security, and reliability from an end-to-end perspective.

Here are some types of system testing with examples:

Functional Testing:

- **Purpose:** Verifies that all features and functionalities of the software work as intended according to the requirements.
- **Example:** In an e-commerce application, functional testing would involve verifying that users can successfully add items to a shopping cart, proceed to checkout, make a payment, and receive an order confirmation.

Performance Testing:

- **Purpose:** Assesses the system's responsiveness, stability, scalability, and resource usage under various load conditions.
- **Example:** For a banking application, performance testing might involve simulating thousands of concurrent users performing transactions to check the system's response time, throughput, and stability under heavy load.

Security Testing:

- **Purpose:** Identifies vulnerabilities and weaknesses in the system that could be exploited by malicious actors.
- **Example:** In a social media platform, security testing would involve attempting to bypass login mechanisms, inject malicious
- code (e.g., SQL injection, XSS), or access unauthorized user data.

Usability Testing:

- **Purpose:** Evaluates how easy and intuitive the software is for end-users to learn and operate.
- **Example:** For a mobile application, usability testing might involve observing real users navigating through the app, performing specific tasks, and gathering feedback on the user interface and overall user experience.

Recovery Testing:

- **Purpose:** Verifies the system's ability to recover from failures, crashes, or data loss.
- **Example:** In a database system, recovery testing would involve simulating a power outage or a server crash and then verifying that the system can restart, restore data integrity, and resume normal operations without data loss.

Regression Testing:

- **Purpose:** Ensures that new changes or bug fixes introduced into the system do not negatively impact existing functionalities.
- **Example:** After a new feature is added to a content management system, regression testing would involve re-running previously passed test cases for existing functionalities (e.g., user login, content creation, page editing) to ensure they still work correctly.

Migration Testing:

- **Purpose:** Verifies a smooth and accurate transition when migrating data or systems from an old environment to a new one.
- **Example:** When upgrading a company's internal software from an older version to a newer one, migration testing ensures that all data is correctly transferred, and the new system functions as expected with the migrated data.

Art of Dubugging

Debugging in software engineering is the systematic process of finding, isolating, and resolving defects or "bugs" in software code. It is an essential skill that transforms a developer's ability to create robust and reliable applications. The "art" of debugging lies in the combination of technical proficiency, logical reasoning, and a methodical approach to problem-solving.

Key Steps in Debugging:

Reproduce the Bug:

The first and most critical step is to consistently replicate the issue. This involves understanding the exact conditions and steps that lead to the bug's occurrence. Without a reproducible bug, identifying the root cause becomes significantly more challenging.

Isolate the Problem:

Once the bug is reproducible, the next step is to narrow down the section of code or component responsible.

Print Statements/Logging: Inserting print statements or using logging frameworks to output variable values, execution flow, and other relevant information at various points in the code.

Using a Debugger: Utilizing an Integrated Development Environment (IDE) debugger to set breakpoints, step through code line by line, inspect variable values, and examine the call stack.

Binary Search Debugging: For larger codebases, commenting out or reverting sections of code to isolate the problematic area.

Identify the Root Cause:

With the problematic area identified, the focus shifts to understanding why the bug is happening. This involves analyzing the code, data flow, and interactions between different components. Common causes include:

- Logical errors
- Incorrect assumptions about data or external systems
- Off-by-one errors in loops or array indexing
- Resource leaks (memory, file handles)
- Race conditions in concurrent programming

Formulate and Implement a Fix:

Once the root cause is understood, a solution is devised and implemented. This might involve correcting a logical flaw, adding error handling, optimizing resource usage, or addressing concurrency issues.

Test the Fix:

After implementing the fix, thorough testing is crucial to ensure that the bug has indeed been resolved and that no new issues (regressions) have been introduced. This often involves running the original steps to reproduce the bug, as well as running relevant unit and integration tests.

Example: Debugging a Python Script

Consider a simple Python script designed to calculate the average of a list of numbers, but it sometimes produces an incorrect result:

Python

```
def calculate_average(numbers):
    total = 0
    for i in range(len(numbers)):
        total += numbers[i]
    return total / len(numbers)
data = [10, 20, 30, 0]
average = calculate_average(data)
print(f"The average is: {average}")
```

Debugging Steps:

Reproduce:

Running this code with data = [10, 20, 30, 0] produces an average of 15.0. If the expected average was 20.0 (excluding the zero), the bug is reproduced.

Isolate:

The issue likely lies within the calculate_average function. We can add print statements to inspect total and len(numbers):

Python

```
def calculate_average(numbers):
    total = 0
    for i in range(len(numbers)):
        total += numbers[i]
    print(f"Total: {total}, Number of elements: {len(numbers)}") # Added print
    return total / len(numbers)
```

This reveals Total: 60, Number of elements: 4. The calculation 60 / 4 correctly yields 15.0. The problem is not in the calculation, but in the input data or the intended logic.

Identify Root Cause: The intended behavior was to calculate the average of non-zero numbers. The presence of 0 in the data list is skewing the average.

Fix: Modify the calculate_average function to exclude zeros:

Python

```
def calculate_average(numbers):
    valid_numbers = [num for num in numbers if num != 0] # Exclude zeros
    if not valid_numbers: # Handle empty list after filtering
        return 0
    total = sum(valid_numbers)
    return total / len(valid_numbers)
```

Test: Rerunning with data = [10, 20, 30, 0] now produces 20.0, which is the expected result. Testing with an empty list or a list containing only zeros would also be prudent.

Case Study

Case Study 1: Library Management System

Related Topic: Design Process and Design Quality

Case Study

A college library plans to replace its manual system with software that manages books, members, issue/return operations, and fine calculations. The software should be easy to maintain, secure, and user-friendly.

Tasks:

Identify the design process followed.

Discuss the quality attributes of the design.

Suggest improvements for better maintainability.

Case Study 2: Online Banking System

Related Topic: Design Concepts and Design Model

Case Study

A bank develops an online banking application that allows customers to transfer funds, pay bills, and view account statements securely.

Tasks:

Identify the design concepts used (abstraction, modularity, information hiding).

Explain how a design model can be prepared for the system.

Case Study 3: Hospital Management System

Related Topic: Software Architecture and Architectural Design

Case Study

A hospital develops software that integrates patient registration, doctor scheduling, laboratory services, pharmacy, and billing.

Tasks:

- Recommend a suitable software architecture.
- Explain how architectural design improves system scalability and maintenance.

Case Study 4: E-Commerce Website

Related Topic: Data Design

Case Study

An online shopping company manages customer accounts, product catalogs, shopping carts, orders, and payments.

Tasks:

Design the major data entities.

Explain how proper data design improves system performance and data consistency.

Case Study 5: Food Delivery Application

Related Topic: Architectural Styles and Patterns

Case Study

A food delivery application connects customers, restaurants, and delivery partners through mobile and web applications.

Tasks:

Identify a suitable architectural style (Layered or Client-Server).

Suggest a suitable design pattern (MVC).

Justify your selection.

Case Study 6: Student Examination System

Related Topic: Test Strategies for Conventional Software

Case Study

A university develops an online examination portal where students attend exams and faculty evaluates answer scripts.

Tasks:

- Identify suitable testing levels.
- Prepare a testing strategy before deployment.
- Explain why integration and system testing are necessary.

Case Study 7: Railway Reservation System

Related Topic: Validation Testing and System Testing

Case Study

A railway reservation system allows users to search trains, reserve tickets, cancel bookings, and make online payments.

Tasks:

Explain how validation testing ensures user requirements are satisfied.

Identify system testing activities before software release.

List possible test cases.

Case Study 8: ATM Banking System

Related Topic: The Art of Debugging

Case Study

After deployment, customers report that ATM transactions occasionally fail after entering the PIN, although the account balance remains unchanged.

Tasks:

- Identify possible software bugs.
- Explain the debugging process to locate the fault.
- Suggest methods to prevent similar errors in future releases.

Text Books

1. *Software Engineering: A Practitioner's Approach* — Roger S. Pressman & Bruce R. Maxim, 9th Edition, McGraw-Hill Education.
2. *Software Engineering* — Ian Sommerville, 10th Edition, Pearson Education.
3. *Fundamentals of Software Engineering* — Rajib Mall, 5th Edition, PHI Learning Pvt. Ltd.
4. *Software Engineering* — K.K. Aggarwal & Yogesh Singh, 3rd Edition, New Age International Publishers.
5. *Software Engineering: Principles and Practice* — Hans van Vliet, 3rd Edition, John Wiley & Sons (Wiley India).

Reference Books

1. *Software Architecture in Practice* — Len Bass, Paul Clements & Rick Kazman, 4th Edition, Addison-Wesley Professional.
2. *Clean Architecture: A Craftsman's Guide to Software Structure and Design* — Robert C. Martin, 1st Edition, Pearson Education.
3. *Design Patterns: Elements of Reusable Object-Oriented Software* — Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides, 1st Edition, Addison-Wesley Professional.
4. *Software Testing: Principles and Practices* — Srinivasan Desikan & Gopalaswamy Ramesh, 2nd Edition, Pearson Education.
5. *Software Testing* — Ron Patton, 2nd Edition, Sams Publishing.
6. *Effective Methods for Software Testing* — William E. Perry, 3rd Edition, Wiley India.
7. *The Art of Software Testing* — Glenford J. Myers, Corey Sandler & Tom Badgett, 3rd Edition, John Wiley & Sons.

Question Number	Questions	Bloom's Level:
Unit – IV : DESIGN ENGINEERING& ARCHITECTURE, TESTING STRATEGIES		
PART – A (2 Marks)		
1	Define the software design process.	L1
2	What is design quality in software engineering?	L1
3	Define software design concepts.	L1
4	What is a design model?	L1
5	Define software architecture.	L1
6	What is architectural design?	L1
7	Define data design.	L1
8	What is an architectural style?	L1
9	What is a software design pattern?	L1
10	Define software testing strategy.	L1
11	What is validation testing?	L1
12	Define system testing.	L1
13	What is debugging?	L1
14	Differentiate between validation testing and system testing.	L2
15	Why is software architecture important in software design?	L2
PART - B(10 Marks)		
1	Ex plain the software design process and discuss the characteristics of a good software design.	L2
2	Describe software design concepts and explain their role in developing quality software.	L2
3	Explain the design model and discuss its major elements with suitable examples.	L2
4	Discuss the process of creating an architectural design and explain the importance of software architecture.	L3
5	Explain the principles of data design and discuss their significance in software development.	L2
6	Compare different architectural styles and patterns with suitable examples.	L4
7	Explain architectural design and discuss the activities involved in developing an effective software architecture.	L3
8	Describe the strategic approach to software testing and explain its objectives.	L2
9	Explain the test strategies for conventional software with appropriate testing levels.	L3
10	Discuss the validation testing process and explain how it ensures software quality.	L3
11	Explain system testing and describe its various types with suitable examples.	L2

Unit V: TESTING TACTICS, SOFTWARE MEASUREMENT AND ESTIMATION

"Testing is the process of executing a program with the intent of finding errors."

— Glenford J. Myers

Unit Overview

This unit introduces the fundamentals of software testing, software metrics, estimation models, and software quality management. It covers white-box and black-box testing techniques, software quality metrics, empirical estimation models, software quality assurance, and formal technical reviews to ensure reliable, efficient, and high-quality software products.

Objectives of the Unit

- Understand the fundamentals of software testing.
- Learn white-box and black-box testing techniques.
- Study software metrics and estimation models.
- Understand software quality measurement methods.
- Learn the principles of software quality assurance.
- Understand the role of formal technical reviews in improving software quality.

Learning Outcomes

After completing this unit, students will be able to:

- Explain software testing fundamentals and testing techniques.
- Apply white-box and black-box testing methods.
- Evaluate software using size-oriented, function-oriented, and quality metrics.
- Apply empirical estimation models for software projects.
- Explain software quality assurance activities.
- Conduct formal technical reviews to improve software quality.

Importance of Studying this Unit

- Improves software reliability and quality.
- Helps identify and remove defects early.
- Supports accurate software cost and effort estimation.
- Enables measurement of software performance and quality.
- Enhances software development and quality assurance practices.
- Reduces maintenance costs through systematic testing and reviews.

Key Concepts

- Software Testing Fundamentals
- White-Box Testing
- Basis Path Testing
- Control Structure Testing
- Black-Box Testing
- Size-Oriented Metrics
- Function-Oriented Metrics
- Software Quality Metrics
- Empirical Estimation Models
- Software Quality Assurance (SQA)
- Formal Technical Reviews (FTR)
- Software Quality Management
- Defect Detection

Software testing fundamentals

Software testing is the fundamental process of verifying and validating a software application to ensure it meets technical and business requirements and is free of defects. It is a critical part of the Software Development Life Cycle (SDLC) that reduces risk, improves product quality, and enhances customer satisfaction.

Importance of software testing

- **Early defect detection**: Finding and fixing bugs in the early stages of development is significantly cheaper and easier than addressing them after the software has been deployed.
- **Improved product quality**: Testing ensures that the software meets quality standards, making it more reliable, efficient, and usable.
- **Customer satisfaction**: A reliable and high-performing application that fulfills user expectations can lead to greater customer satisfaction and brand loyalty.
- **Security**: systematic testing uncovers security vulnerabilities and loopholes, protecting user data and preventing malicious attacks.
- **Cost-effectiveness**: While testing requires resources, the cost of fixing post-release failures is often much higher than the investment in early testing.

Software Testing Life Cycle (STLC)

The STLC is a structured sequence of activities that ensures a systematic approach to testing. The typical phases include:

- **Requirement Analysis**: The testing team analyzes the software requirements to identify testable features, prioritize testing efforts, and prepare a Requirement Traceability Matrix (RTM).

- **Test Planning**: Test managers create a test plan document that defines the testing scope, strategy, tools, resources, and timelines.
- **Test Case Development**: Testers write detailed test cases, prepare test data, and review them for accuracy and traceability to requirements.
- **Test Environment Setup**: The hardware, software, and network are configured to take off the production environment. A smoke test is often run to verify the setup.
- **Test Execution**: Test cases are executed, and results are logged. Any deviations from expected behavior are reported as defects to the development team.
- **Test Cycle Closure**: The testing team analyzes the test results, prepares a test summary report, and documents lessons learned for future projects.

Core testing types with examples

Software testing can be categorized by the underlying methodology and by what is being tested.

Based on test approach

- **Manual Testing**: Performed by human testers who manually execute test cases without automation tools. It is flexible and good for exploratory testing.
    - **Example**: A tester manually enters a username and password into a login form to verify that they can access their account.
- **Automated Testing**: Uses scripts and tools to execute repetitive test cases quickly and efficiently, especially useful for large systems and regression testing.
    - **Example**: A script is created using a tool like Selenium to automatically test that all the links on a website's navigation bar are functional.

White box texting

White box testing is a software testing method that focuses on the internal structure, logic, and code of a software application. Unlike black box testing, which treats the software as a sealed system, white box testing requires knowledge of the inner workings to design test cases and scrutinize the internal operations.

It is also known by other names, including structural, clear box, glass box, or transparent box testing.

Fundamentals of white box testing

- **Access to source code:** Testers have complete access to the internal code, design documents, and architecture of the application. This allows them to create tests based on the actual implementation rather than just the stated requirements.
- **Code coverage analysis:** A primary goal is to ensure that a high percentage of the code is executed by the test cases. Techniques are used to measure and maximize code coverage, including statement, branch, and path coverage.
- **Verification of internal logic:** The main focus is on verifying that the program's logic and control flow are correct. This includes checking conditional statements (e.g., if-else), loops, and internal functions.
- **Early bug detection:** White box testing is often performed early in the development cycle, typically by developers during unit testing. This allows for the detection and correction of bugs at a stage when they are less expensive and easier to fix.
- **Requires programming skills:** Testers must possess programming knowledge to understand the code, identify testable paths, and write the necessary test cases.

Key techniques and examples

1. Statement coverage

This technique ensures that every executable statement or line of code is tested at least once. It verifies that no line of code is "dead" or unused.

Example

Consider the following pseudo-code for a function that checks if a user is eligible for a senior discount:

```
function check_senior_discount(age, has_loyalty_card):
  discount_eligible = false
  if age >= 65:
    discount_eligible = true
  if has_loyalty_card:
    discount_eligible = true
  return discount_eligible
```

To achieve 100% statement coverage, you need two test cases:

- **Test Case 1:** An input that executes the first if statement (e.g., age = 70, has_loyalty_card = false).
- **Test Case 2:** An input that executes the second if statement (e.g., age = 45, has_loyalty_card = true).

2. Branch coverage (or decision coverage)

This technique ensures that every branch (true and false outcomes) of each decision point in the code is executed at least once.

Example

Using the same check_senior_discount function:

```
function check_senior_discount(age, has_loyalty_card):
  discount_eligible = false
  if age >= 65:
    discount_eligible = true
  if has_loyalty_card:
    discount_eligible = true
  return discount_eligible
```

To achieve branch coverage, you would need four test cases to cover all combinations of the two conditions:

Test Case 1: age = 70, has_loyalty_card = true (both conditions are true).

Test Case 2: age = 70, has_loyalty_card = false (first condition true, second false).

Test Case 3: age = 45, has_loyalty_card = true (first condition false, second true).

Test Case 4: age = 45, has_loyalty_card = false (both conditions are false).

3. Loop testing

This technique focuses specifically on validating the correct behavior of loops within the code. Test cases are designed to test the loop for zero, one, and multiple iterations, as well as for boundary conditions.

Example
Consider a function that calculates the sum of a list of numbers:

```
function calculate_sum(numbers):
  sum = 0
  for number in numbers:
    sum += number
  return sum
```

Test cases for loop testing would include:

- **Zero iterations:** Pass an empty list []. The expected output is 0.
- **One iteration:** Pass a list with one item [5]. The expected output is 5.
- **Multiple iterations:** Pass a list with several items [1, 2, 3]. The expected output is 6.

Basis Path Testing

Basis path testing is a white-box testing technique in software engineering that aims to achieve thorough code coverage by testing a set of linearly independent paths through a program's control flow. It focuses on the internal structure of the code, making it a valuable tool for developers at the unit level.

Fundamentals of Basis Path Testing:

Control Flow Graph (CFG) Construction:

The first step involves creating a graphical representation of the program's control flow. This graph consists of nodes (representing blocks of code) and edges

(representing the flow of control between these blocks). Decision points (e.g., if statements, while loops) lead to multiple outgoing edges.

Cyclomatic Complexity Calculation:

This metric, often attributed to McCabe, quantifies the logical complexity of a program and determines the minimum number of independent paths that must be tested. It can be calculated using various formulas, such as $V(G) = E - N + 2P$ (where E is the number of edges, N is the number of nodes, and P is the number of connected components) or by counting regions in the CFG.

Identification of Independent Paths:

Based on the cyclomatic complexity, a basis set of independent paths is identified. Each path in this set introduces at least one new edge or decision that has not been covered by previous paths.

Test Case Design:

Test cases are then designed to execute each of these independent paths. This involves setting up specific input conditions and data that will force the program to traverse the intended path through the CFG.

Example:

Consider the following simple code snippet:
Python

```
def calculate_grade(score):
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    else:
        return "C"
```

Steps for Basis Path Testing:

- **Control Flow Graph:**
  - Node 1: def calculate_grade(score):
  - Node 2: if score >= 90:
  - Node 3: return "A"
  - Node 4: elif score >= 80:
  - Node 5: return "B"
  - Node 6: else:
  - Node 7: return "C"

Edges connect these nodes based on the control flow. For example, from Node 2, there are edges to Node 3 (if true) and Node 4 (if false).

Cyclomatic Complexity:

Using the formula V(G) = E - N + 2P, or by counting regions, the cyclomatic complexity for this example would be 3 (representing the three distinct grade paths).

Independent Paths:

- Path 1: Node 1 -> Node 2 (true) -> Node 3 (Score >= 90)
- Path 2: Node 1 -> Node 2 (false) -> Node 4 (true) -> Node 5 (80 <= Score < 90)
- Path 3: Node 1 -> Node 2 (false) -> Node 4 (false) -> Node 6 -> Node 7 (Score < 80)

Test Cases:

- Test Case 1: calculate_grade(95) (Expected: "A")
- Test Case 2: calculate_grade(85) (Expected: "B")
- Test Case 3: calculate_grade(75) (Expected: "C")

By designing test cases for each independent path, basis path testing ensures that all logical conditions and branches within the code are exercised, leading to improved code coverage and the detection of potential errors.

Control structure testing

Control structure testing is a white-box testing technique that validates a program's internal logic and flow by testing its decision statements, loops, and data flows. It is a fundamental part of structural testing and requires knowledge of the source code and its internal implementation.

Key Techniques and Examples

The main techniques used in control structure testing include Condition Testing, Data Flow Testing, and Loop Testing.

1. Condition Testing

Condition testing focuses on ensuring that the logical conditions and decision-making statements (like if, else, switch) in a program are free from errors. Test cases are designed to make each condition evaluate to both true and false outcomes.

Example:
Consider the following pseudo-code for a discount eligibility check:

```
IF (age > 60 AND is_member == true) THEN
apply_discount = true
ELSE
apply_discount = false
END IF
```

To test this using condition testing (specifically, multiple-condition coverage), test cases should cover all combinations of the simple conditions age > 60 and is_member == true:

- **Test Case 1:** age = 65, is_member = true (Both true, apply_discount = true)
- **Test Case 2:** age = 65, is_member = false (One true, one false, apply_discount = false)
- **Test Case 3:** age = 55, is_member = true (One false, one true, apply_discount = false)
- **Test Case 4:** age = 55, is_member = false (Both false, apply_discount = false)

2. Loop Testing

Loop testing is a white-box technique that specifically targets the validity of loop constructs (for, while, do-while). Errors often occur at loop boundaries, so test cases focus on the number of iterations.

Example:

Consider a simple for loop that iterates n times, where n is the maximum allowable number of passes.

```
FOR i from 1 to n:
   perform_action()
END FOR
```

Test cases for this loop should include:

- **Skip the loop entirely** (e.g., set n=0 or make the condition false initially).
- **Traverse the loop only once** (e.g., set n=1).
- **Traverse the loop two times** (e.g., set n=2).
- **Traverse the loop m times** where m < n (e.g., set n=10, m=5).
- **Traverse the loop n-1, n, and n+1 times** (e.g., test the boundaries).

3. Data Flow Testing

Data flow testing selects test paths based on the locations where variables are defined and used (definition-use chains) within the program. The goal is to ensure variables are handled correctly and to uncover anomalies like uninitialized variables or defined variables that are never used.

Example:

Consider a code segment where variable X is defined in statement S and used later in statement S'. A test case would be designed to follow a path from S to S' without any redefinition of X in between to ensure the value flows correctly. This helps to identify issues such as if the calculation for X was skipped under certain conditions.

Black box testing

Black box testing is a software testing method in which the tester evaluates an application's external functionality and behavior without any knowledge of its internal code structure, implementation details, or design. The tester interacts with the software through its user interface or APIs, provides inputs, and verifies that the outputs match the expected results based on the software's requirements and specifications.

Key Concepts

- **Focus on functionality:** The primary goal is to ensure the software performs its intended tasks correctly from the end-user's point of view.
- **No internal knowledge required:** Testers do not need programming skills or access to the source code, which allows for an objective and unbiased assessment.
- **Requirements-based:** Test cases are created based on the software's functional and non-functional requirements documents.

Example

Consider testing the **login page** of a banking website.

1. **Identify Functionality:** The main function is to allow users to log in with valid credentials and reject invalid ones.

2. **Create Test Cases:**

Valid Input: Enter a correct username and a correct password.

- *Expected Result:* The user is successfully logged in and redirected to their account dashboard.

Invalid Username: Enter an incorrect username and a correct password.

- *Expected Result:* An error message is displayed: "Invalid username or password".

Invalid Password: Enter a correct username and an incorrect password.

- *Expected Result:* An error message is displayed, and the number of failed attempts is tracked (e.g., after a certain number of attempts, the account is locked).

Boundary Conditions: Leave the username and/or password fields blank.

- *Expected Result:* An error message prompting the user to enter credentials appears.

Execute and Verify: The tester runs these scenarios and compares the actual outcome with the expected outcome. If they match, the test case passes. If not, a bug is reported to the development team.

Common Techniques

Testers use specific techniques to reduce the number of test cases while ensuring good coverage:

- **Equivalence Partitioning:** Dividing input data into distinct "classes" or "partitions" where all values within a class are expected to produce the same result. Only one value from each class is tested.
- **Boundary Value Analysis (BVA):** Testing values at the edges of valid and invalid input ranges (e.g., for an age input field accepting values 18-30, testers would test 17, 18, 30, and 31).
- **Decision Table Testing:** Used for systems with complex business rules where the output depends on multiple combinations of inputs.

- **State Transition Testing:** Focusing on how the system changes from one state to another in response to specific events or inputs (e.g., an account locking after multiple failed login attempts).
- **Error Guessing:** Using the tester's experience and intuition to anticipate where defects might occur and designing test cases accordingly.

Size Oriented Metrics

Size-oriented metrics are derived by normalizing quality and productivity Point Metrics measures by considering the size of the software that has been produced. The organization builds a simple record of size measure for the software projects. It is built on past experiences of organizations. It is a direct measure of software. This metric **measure** is one of the simplest and earliest metrics that is used for computer programs to measure size.

Size Oriented Metrics are also used for measuring and comparing the productivity of programmers. It is a direct measure of a Software. The size measurement is based on lines of code computation. The lines of code are defined as one line of text in a source file. While counting lines of code, the simplest standard is:

- Don’t count blank lines
- Don’t count comments
- Count everything else
- The size-oriented measure is not a universally accepted method.

A simple set of size measures that can be developed is given below:

Size = Kilo Lines of Code (KLOC)
Effort = Person / month
Productivity = KLOC / person-month
Quality = Number of faults / KLOC
Cost = $ / KLOC
Documentation = Pages of documentation / KLOC

Advantages of Size-Oriented Metrics

- **Easy to Use:** Simple to apply by counting things like lines of code or function points, making it quick to estimate effort and costs.
- **Early Estimation:** Helps predict the size of a project during the planning phase, allowing for realistic budgets and timelines before coding starts.
- **Easy Comparison Across Projects:** Standardized metrics make it easy to compare projects, aiding in benchmarking and cost prediction.
- **Builds Historical Data:** Tracks past project data (like cost per line of code), improving the accuracy of future estimates.
- **Reduces Guesswork:** Uses concrete measurements instead of guesses, leading to more consistent and reliable results.
- **Great for Large Projects:** Breaks large projects into manageable chunks, simplifying progress tracking and cost management.

Example of Size-Oriented Metrics

For a size oriented metrics, software organization maintains records in tabular form. The typical table entries are: Project Name, LOC, Efforts, Pages of documents, Errors, Defects, Total number of people working on it.

Project Name	LOC	Effort	Cost ($)	Doc. (pages)	Errors	Defects	People
ABC	10, 000	20	170	400	100	12	4
PQR	20, 000	60	300	1000	129	32	6
XYZ	20, 000	65	522	1290	280	87	7

Function-oriented metrics

Function-oriented metrics use a measure of the functionality delivered by the application as a normalization value, focusing on the software's behavior and user-visible services rather than implementation details like lines of code. The most widely used example is **Function Point (FP) analysis**.

Core Concepts

- **User-centric:** These metrics measure software characteristics from the user's perspective, based on what the user requests and receives.
- **Language-independent:** They are not tied to a specific programming language, making them useful for comparing projects using different technologies.
- **Early Estimation:** Since they rely on functional requirements, they can be calculated earlier in the project lifecycle than code-based metrics, aiding in initial effort and cost estimations.

Measurement Parameter	Definition
External Inputs (EI)	Each unique user input type that provides distinct application-oriented data to the software (e.g., a data entry screen or form).
External Outputs (EO)	Each unique output type that provides application-oriented information to the user (e.g., reports, error messages, or screens).
External Inquiries (EQ)	Each unique on-line input that results in the generation of an immediate on-line output/response (e.g., a search query).
Internal Logical Files (ILF)	A logical grouping of data or a master file maintained within the application's boundary (e.g., a database table).
External Interface Files (EIF)	A logical grouping of data that is referenced by the application but maintained by another external application (e.g., an interface file from another system).

Primary Example: Function Points (FP) Analysis

The Function Point (FP) metric, developed by Albrecht at IBM, is the primary function-oriented metric. It quantifies the software's size by counting and weighting five key information domain characteristics:

Calculation Steps (Simplified)

1. **Count** the occurrences of each parameter (EI, EO, EQ, ILF, EIF).
2. **Assign complexity** (simple, average, or complex) to each count based on established criteria.
3. **Apply weighting factors** to these counts to get an initial function point total.
4. **Adjust for overall system complexity** using 14 global adjustment factors (e.g., data communications, performance criticality, reusability) to arrive at the final Function Point count.

Other Examples of Function-Oriented Metrics

- **Feature Points:** An extension of function points, often used for systems with significant internal processing rather than user interaction, such as operating systems or real-time software.
- **Use Case Points (UCP):** Used in agile methodologies, UCPs measure the system's size based on the number and complexity of use cases and actors involved.
- **Object Points (OP):** Measures size based on the number and complexity of objects, attributes, and methods defined in object-oriented design.

Metrics for Software quality

Software quality metrics include **defect density** (defects per lines of code), **test coverage** (percentage of code covered by tests), and **Mean Time to Recovery (MTTR)** (average time to restore service after a failure). Other key metrics are **lead time for changes** (time from commit to deployment), **change failure rate** (percentage of deployments causing failures), and **customer-reported bugs** (number of unique defects reported by users).

Product quality metrics

- **Defect Density:** Measures the number of defects per unit of size (e.g., lines of code or function points).
    - **Example:** 10 defects in 20,000 lines of code results in a defect density of 0.5 defects per 1,000 lines of code.
- **Mean Time to Failure (MTTF)/Mean Time Between Failures:** The average time between one failure and the next.
    - **Example:** Used for safety-critical systems like avionics; an MTTF of 500 hours means the system is expected to fail, on average, every 500 hours of use.
- **Customer-Reported Bugs:** The number of unique defects reported by customers, which can be further analyzed by severity.
    - **Example:** 50 bugs were reported by users in the last quarter.
- **Defect Leakage:** The number of defects that escape the development team and are found by the customer or during user acceptance testing (UAT).
    - **Example:** 5 critical bugs were found in production that should have been caught during internal testing.

Process and performance metrics

- **Test Coverage:** The percentage of your codebase that is executed by your automated tests.
    - **Example:** A test coverage of 85% means that 85% of the application's code is being tested.

- **Lead Time for Changes:** The time it takes from a code commit to that code successfully running in production.
    - **Example:** It takes an average of 2 hours for a code change to go from commit to deployment.
- **Mean Time to Recovery (MTTR):** The average time it takes to restore service after a system failure.
    - **Example:** A server fails, and it takes, on average, 15 minutes to bring it back online.
- **Change Failure Rate:** The percentage of deployments that result in a failure and require remediation (e.g., a rollback or hotfix).
    - **Example:** Out of 100 deployments, 5 resulted in a failure, giving a 5% change failure rate.
- **Code Churn:** The rate at which code is being modified, added, or deleted over a specific period.
    - **Example:** A specific module of code has a high churn rate, indicating it is unstable and potentially error-prone.

Empirical Estimation Models

Empirical estimation models are mathematical formulas used in software engineering to predict the effort, cost, and duration of a project, based on historical data and observed metrics. These models are more reliable than guesswork, providing a data-driven foundation for project planning and risk management.

Core concepts of empirical estimation

- **Historical data:** The models are built on data collected from past projects, including metrics such as project size, team experience, development tools, and software complexity.

- **Statistical analysis:** Statistical techniques like regression analysis are used to analyze the historical data. This helps determine the relationship between project variables and outcomes, allowing for the derivation of predictive formulas.
- **Effort and size relationship:** The core principle is that a relationship exists between the size of a software product and the effort required to develop it. This is typically an exponential relationship, where effort increases at an accelerating rate as project size grows.
- **Customization:** No single model fits every project or organization. The constants within the formulas must be calibrated and adjusted to reflect a specific organization's environment, processes, and historical data.
- **Project drivers:** Sophisticated models incorporate "cost drivers" or "effort multipliers," which are characteristics of the project that can influence the final effort. These factors include product reliability, database size, and programmer capability.

Examples of empirical estimation models

1. Constructive Cost Model (COCOMO)

Developed by Barry Boehm, COCOMO is one of the most widely known empirical estimation models. It has evolved into several versions, including COCOMO II, to accommodate modern software practices.

The formula: The basic COCOMO formula for estimating effort (in person-months) is:

1. $Effort = a \times (Size)b$
2. $Size$ is measured in thousands of lines of code (KLOC).
3. a and b are constants determined by the project's development mode

Development modes: The original COCOMO model classifies projects into three modes, each with different values for a and b:

- **Organic:** Small, simple projects developed by small, experienced teams with flexible requirements.
- **Semi-detached:** Intermediate projects with a mix of team experience and requirements.

- **Embedded:** Complex, large-scale projects with tight hardware, software, and operational constraints.

Example of Basic COCOMO:

A semi-detached project is estimated to have a size of 50,000 lines of code (50 KLOC). For this mode, the constants area a=3.0 and b=1.12.

$Effort$=3.0×(50)1.12≈202 person-months.

- This suggests the project would require approximately 202 person-months of effort.

2. The Software Equation

This is a dynamic, multivariable model that uses a formula derived from data

The formula: ıution of

$$E = B \times (Size/P)^{t^3/333}$$

- E = Effort (in person-months or person-years)
- t = Project duration (in months or years)
- B = A special skills factor (0.16 to 0.39)
- P = A productivity parameter reflecting development practices (with typical values ranging from 2,000 for embedded software to 28,000 for business applications).

Example of the Software Equation:

A business application with an estimated size of 100 KLOC is being developed.

- Assume the duration (t) is 12 months, the skills factor (B) is 0.20, and the productivity parameter (P) is 28,000.
- $Effort = 0.20 \times (100000/28000)^{12^3/333} \approx 13$ person-months.
- This model can also be rearranged to estimate project duration based on effort, or size

Software Quality Management

Software Quality Management (SQM) is a systematic process of ensuring that software products meet or exceed customer expectations and regulatory requirements. It is integrated throughout the entire Software Development Life Cycle (SDLC) to build quality into the product from the start rather than just testing for it at the end.

Core SQM concepts

SQM is composed of three interconnected techniques: quality assurance, quality planning, and quality control.

1. Quality Assurance (QA)

This is a proactive, process-oriented approach focused on preventing defects. QA ensures the development processes themselves are sound, so that the likelihood of producing a low-quality product is minimized.

- **Concept:** Establishing and implementing reliable processes, standards, and methodologies that the entire team follows.
- **Example:** A software company implements a policy requiring all code to pass a static code analysis tool (like SonarQube) and be reviewed by at least one other developer before it can be merged into the main branch. This prevents common coding errors and inconsistencies.

2. Quality Planning (QP)

This involves defining the specific quality objectives and standards for a particular project. It outlines the strategy for how the team will achieve, measure, and control quality throughout the development process.

- **Concept:** Proactively defining quality goals, metrics, and the resources and processes needed to achieve them. This is often documented in a Quality Management Plan.
- **Example:** For a new banking app, the quality plan defines specific metrics such as a target response time for transactions (e.g., under 2 seconds) and a required code test coverage of 80%. It also specifies that user acceptance testing (UAT) will be performed by a representative group of customers before launch.

3. Quality Control (QC)

This is a reactive, product-oriented process that focuses on identifying and correcting defects in the final product. QC activities like testing are performed to verify that the software meets the quality standards established during the planning and assurance phases.

- **Concept:** The set of operational techniques and activities used to check for conformance to quality standards and remove defects.
- **Example:** The QA team executes test cases defined in the quality plan to find bugs and defects. They perform functional tests to ensure all features work as specified and load tests to verify the system can handle expected user traffic. Any defects found are logged in a bug-tracking system like Jira, and the team works to resolve them.

Key concepts across the SDLC

These core concepts are applied throughout the software development life cycle (SDLC) to ensure continuous quality improvement.

Metrics for measuring quality

Metrics provide objective data to track the effectiveness of SQM efforts.

- **Defect Density:** The number of confirmed defects per unit of code (e.g., per thousand lines of code).
    - **Example:** After a feature is tested, the team calculates the defect density. If a new module has a much higher defect density than older, stable modules, it signals a potential quality problem that requires further attention.
- **Mean Time to Recovery (MTTR):** The average time it takes to restore service after a production failure or incident.
    - **Example:** The operations team tracks MTTR to measure the team's incident response effectiveness. A low MTTR indicates that the team can quickly diagnose and fix critical issues in a production environment.

- **Test Coverage:** The percentage of code that is executed by tests.
    - **Example:** A company uses a tool to measure its unit test coverage, finding it is only 50%. The team decides to increase this to 80% to ensure better code quality and fewer defects.

Software Quality Assurance

Software Quality Assurance (SQA) is a systematic process of ensuring that a software product meets its specified quality standards and requirements. It is a proactive, process-oriented discipline that works in parallel with the software development lifecycle to prevent defects from occurring in the first place, in contrast to Quality Control (QC), which is a reactive process focused on finding defects in the finished product.

Core concepts of software quality assurance

1. Process-oriented approach

SQA focuses on improving the development process itself rather than just inspecting the final product. By establishing and refining repeatable, high-quality processes, the likelihood of defects and errors is significantly reduced.

- **Example:** A company implements the Agile methodology, using short development cycles ("sprints"). A key SQA activity is to conduct a process audit at the end of each sprint to identify inefficiencies and improve the workflow. This might reveal that a specific type of bug frequently arises during unit testing, leading the team to add an extra peer review step for that code module.

2. Defect prevention

This is a core principle of SQA, aiming to prevent problems from entering the software early in the development cycle. By identifying and addressing potential issues at their root cause, costs and development time are significantly reduced.

- **Example:** During the requirements analysis phase, an SQA team reviews the software requirements specification (SRS) document with developers and stakeholders. They identify an ambiguous requirement for handling "invalid user input." To prevent future bugs, the team clarifies the requirement and documents specific validation rules, like setting a maximum character limit and specifying the type of data to be accepted.

3. Continuous improvement

SQA is not a one-time activity but an ongoing process of monitoring, assessing, and refining the software development process. Organizations use models like the Capability Maturity Model Integration (CMMI) to gauge and improve the maturity of their processes.

- **Example:** An SQA team tracks and analyzes bug reports and customer feedback after a new software release. They discover a recurring pattern of usability issues, suggesting the initial design review process was flawed. The team then updates the SQA plan to incorporate usability testing earlier in the design phase, establishing a cycle of continuous improvement.

4. Quality attributes (or factors)

Software quality is defined by a set of measurable characteristics. SQA focuses on ensuring these attributes are achieved throughout the development process. Key attributes include:

- **Reliability:** The software consistently performs as expected without failures.
    - **Example:** An e-commerce application is tested for reliability by simulating high traffic during a major sale event to ensure it doesn't crash or slow down under heavy load.
- **Usability:** The software is easy for users to learn and operate.
    - **Example:** A new mobile banking app is given to a group of first-time users to test the intuitiveness of its navigation and features. Their feedback is used to refine the user interface.
- **Maintainability:** The software can be easily modified, updated, and extended.
    - **Example:** A code inspection is performed to ensure all developers follow the same coding standards and that the code is well-documented. This makes it easier for future developers to understand and modify the code without introducing new errors.
- **Performance efficiency:** The software uses resources like CPU time and memory effectively.

    - **Example:** The development team runs stress tests on a database to see how fast it can retrieve information when handling a large volume of requests. This ensures optimal performance as the user base grows.

- **Security:** The software protects against unauthorized access and data breaches.
    - **Example:** A financial application undergoes regular penetration testing, where ethical hackers attempt to exploit vulnerabilities. This identifies security flaws before malicious actors can.

5. Verification and validation (V&V)

V&V are distinct but complementary activities within SQA.

- **Verification:** Checks if the product is being built correctly. This is often done through reviews, inspections, and walk-throughs to ensure each phase of the development meets the specified requirements.
    - **Example:** During a design review, the SQA team compares the system architecture design against the initial requirements document to verify that all functional needs are being addressed by the design.
- **Validation:** Confirms that the right product was built. It involves testing the software against user expectations and requirements to ensure it meets its intended purpose.
    - **Example:** User Acceptance Testing (UAT) is performed, where end-users test the final product to ensure it meets their expectations and is ready for release.

Formal Technical Reviews (FTRs)

Formal Technical Reviews (FTRs) are a structured and methodical software quality control activity performed by a team of peers to identify defects in software engineering work products. By catching errors early, FTRs prevent defects from advancing to later stages where they become significantly more expensive and difficult to fix.

Core concepts of FTRs

The main goal of FTR is to ensure the quality of a software product by finding and removing errors and inconsistencies. Key objectives include:

- **Defect identification:** To uncover errors in functions, logic, or implementation within any software representation, such as design documents or code.
- **Quality assurance:** To confirm compliance with project specifications, requirements, and established standards.
- **Knowledge sharing:** To promote collaboration and a common understanding among team members, which can also train junior engineers.
- **Risk mitigation:** To proactively identify and manage potential issues that could worsen into major risks later in the project.
- **Cost reduction:** To find and fix defects early, which is much cheaper than correcting them during testing or after deployment.

Key principles

For an FTR to be successful, participants must adhere to a core set of guidelines:

- **Review the product, not the producer:** The review should focus on the quality of the work product, not on the individual who created it. The goal is to be constructive, not critical.
- **Set and maintain an agenda:** A structured agenda ensures the meeting stays focused and on track.
- **Limit debate:** The meeting's purpose is to find problems, not solve them. Solutions are discussed and implemented later.
- **Take written notes:** A designated scribe records all issues and points of discussion for follow-up.
- **Insist on advance preparation:** All participants must review the material beforehand to make the meeting productive.
- **Use a checklist:** A prepared checklist helps to focus the reviewers' attention on important issues and ensures consistency.

Participant roles

An FTR involves a small, focused team (typically 3–5 people) with defined roles.

- **Producer (Author):** The person who created the work product. Their role is to clarify points and address the identified defects after the meeting.
- **Moderator:** The facilitator who leads the FTR meeting. They are responsible for keeping the meeting on track, enforcing the rules, and ensuring objectives are met.
- **Reviewers:** The peers and subject matter experts who examine the work product and identify defects.

Text Books

1. *Software Engineering: A Practitioner's Approach* — Roger S. Pressman & Bruce R. Maxim, 9th Edition, McGraw-Hill Education.
2. *Software Engineering* — Ian Sommerville, 10th Edition, Pearson Education.
3. *Fundamentals of Software Engineering* — Rajib Mall, 5th Edition, PHI Learning Pvt. Ltd.
4. *Software Engineering* — K.K. Aggarwal & Yogesh Singh, 3rd Edition, New Age International Publishers.
5. *Software Testing: Principles and Practices* — Srinivasan Desikan & Gopalaswamy Ramesh, 2nd Edition, Pearson Education.

Reference Books

1. *The Art of Software Testing* — Glenford J. Myers, Corey Sandler & Tom Badgett, 3rd Edition, John Wiley & Sons.
2. *Introduction to Software Testing* — Paul Ammann & Jeff Offutt, 2nd Edition, Cambridge University Press.
3. *Effective Methods for Software Testing* — William E. Perry, 3rd Edition, Wiley India.
4. *Software Testing* — Ron Patton, 2nd Edition, Sams Publishing.
5. *Managing the Software Process* — Watts S. Humphrey, 1st Edition, Addison-Wesley Professional.
6. *Software Metrics: A Rigorous and Practical Approach* — Norman E. Fenton & James Bieman, 3rd Edition, CRC Press.
7. *Practical Software Measurement: Objective Information for Decision Makers* — John C. McGarry, David A. Card, Cheryl L. Jones, Beth A. Layman, Elizabeth M. Clark, Joseph S. Dean & Fred E. Hall, 1st Edition, Addison-Wesley Professional.

Case Study 1: Online Banking System

Related Topic: Software Testing Fundamentals

Case Study

A bank develops an online banking application that allows customers to transfer funds, check balances, and pay utility bills.

Tasks:

- Identify the objectives of software testing.
- List the testing activities before deployment.
- Explain why testing is essential.

Case Study 2: Student Result Management System

Related Topic: White-Box Testing

Case Study

A university develops software to calculate students' grades based on internal and external marks.

Tasks:

- Identify the program logic to be tested.
- Explain how white-box testing ensures complete code coverage.
- Suggest suitable white-box testing techniques.

Case Study 3: ATM Cash Withdrawal System

Related Topic: Basis Path Testing

Case Study

An ATM software validates the PIN, checks account balance, dispenses cash, and updates the account.

Tasks:

- Draw the control flow of the withdrawal process.
- Identify independent execution paths.
- Explain how basis path testing improves software reliability.

Case Study 4: Railway Reservation System

Related Topic: Control Structure Testing

Case Study

A railway reservation system validates passenger details, seat availability, payment, and ticket confirmation.

Tasks:

- Identify decision and loop structures.
- Explain how control structure testing verifies the program logic.
- Suggest test cases.

Case Study 5: Online Shopping System

- Related Topic: Black-Box Testing

Case Study

An e-commerce website allows customers to search products, add items to the cart, make payments, and track orders.

Tasks:

- Identify input and output test cases.
- Explain black-box testing techniques.
- Validate customer requirements through testing.

Case Study 6: Hospital Management System

- Related Topic: Size-Oriented Metrics and Function-Oriented Metrics

Case Study

A hospital develops software to manage patients, doctors, appointments, billing, and medical records.

Tasks:

- Estimate software size using LOC.
- Calculate function points.
- Compare size-oriented and function-oriented metrics.

Case Study 7: Library Management System

Related Topic: Software Quality Metrics

Case Study

A college library develops software for book issue, return, fine calculation, and online reservation.

Tasks:

- Identify software quality metrics.
- Explain how reliability and maintainability can be measured.

Suggest methods to improve software quality.

Case Study 8: Food Delivery Application

Related Topic: Empirical Estimation Models

Case Study

A startup develops a food delivery application with customer, restaurant, and delivery modules.

Tasks:

- Estimate project effort using an empirical estimation model.
- Identify factors affecting project cost.
- Discuss the importance of software estimation.

Case Study 9: College Admission Portal

Related Topic: Software Quality Assurance (SQA)

Case Study

A university develops an online admission portal for application submission, fee payment, and seat allocation.

Tasks:

- Identify SQA activities.
- Explain how quality assurance ensures software reliability.
- Suggest quality standards for the project.

Case Study 10: Airline Reservation System

Related Topic: Formal Technical Reviews (FTR)

Case Study

An airline reservation system is nearing completion. Before deployment, the development team conducts reviews of the design documents, source code, and test reports.

Tasks:

- Explain the purpose of formal technical reviews.
- Identify the participants involved in the review.
- Discuss the benefits of FTR in improving software quality.

UNIT - V: TESTING TACTICS, SOFTWARE MEASUREMENT AND ESTIMATION		
PART – A (2 Marks)		
1	Define software testing.	L1
2	What is white-box testing?	L1
3	Define basis path testing.	L1
4	What is control structure testing?	L1
5	Define black-box testing.	L1
6	What are size-oriented metrics?	L1
7	Define function-oriented metrics.	L1
8	What are software quality metrics?	L1
9	Define empirical estimation models.	L1
10	What is Software Quality Assurance (SQA)?	L1
11	Define Formal Technical Review (FTR).	L1
12	Differentiate between white-box testing and black-box testing.	L2
13	State any two objectives of software quality assurance.	L2
14	Why are software metrics important in software engineering?	L2
PART – B(10 Marks)		
1	Explain the fundamentals of software testing and discuss its objectives and principles.	L2
2	Describe white-box testing and explain its advantages and limitations with suitable examples.	L2
3	Explain basis path testing with the steps involved in identifying independent paths.	L3
4	Discuss control structure testing and explain its different testing techniques.	L3
5	Explain black-box testing and discuss the commonly used black-box testing techniques.	L2
6	Compare white-box testing and black-box testing with suitable examples.	L4
7	Explain size-oriented metrics and discuss their role in software project estimation.	L2
8	Describe function-oriented metrics and explain how function points are used to measure software size.	L3
9	Discuss software quality metrics and explain how they are used to evaluate software quality.	L3
10	Explain empirical estimation models and discuss their importance in software cost and effort estimation.	L2
11	Describe Software Quality Assurance (SQA) and explain its activities in the software development life cycle.	L2