BUDDHA SERIES

(Unit Wise Solved Question & Answers)

Course – B.Tech. (CSE)

College – Buddha Institute of Technology

(AKTU CODE-525)

Department: Computer Science & Engineering(AIML/DS)

Subject: Software Engineering (BCS 601)

Faculty Name: Mr. Satish Kumar

Unit - 1

Q.1 What is the principal aim of the software engineering discipline? **(AKTU 2018-19)**

Ans: The principal aim of the software engineering discipline is to design, develop, maintain, and manage software systems in a systematic, disciplined, and quantifiable manner. This involves applying engineering principles to software creation and ensuring that the software is reliable, efficient, maintainable, and meets the requirements of users and stakeholders.

Key objectives include:

Quality Assurance: Ensuring the software meets specified standards and performs reliably and securely under various conditions.

Efficiency: Developing software that performs well in terms of speed, resource utilization, and scalability.

Maintainability: Creating software that is easy to modify and extend, allowing for future updates and improvements.

Cost-effectiveness: Balancing the trade-offs between time, budget, and feature set to deliver a product that provides maximum value.

User Satisfaction: Ensuring the software meets the needs and expectations of its users, providing a positive user experience.

Risk Management: Identifying, assessing, and mitigating risks associated with software development and deployment.

By focusing on these goals, software engineering aims to produce software that is functional, reliable, and aligned with user and business needs while managing the complexities and challenges inherent in software development.

Q.2 What the different characteristics of software engineering? **(AKTU 2021-22)**

Ans: Software engineering is characterized by several distinct attributes that distinguish it from other engineering disciplines and general software development practices. These characteristics include:

Systematic Approach: Software engineering uses structured methodologies and processes to ensure software development is planned, managed, and executed in an organized manner.

Engineering Principles: It applies principles from traditional engineering disciplines, such as design, analysis, and testing, to software development, ensuring rigor and precision.

Quality Focus: Emphasis is placed on ensuring software quality through practices such as testing, code reviews, and quality assurance processes to produce reliable, robust, and secure software.

Lifecycle Management: Involves managing the entire software development lifecycle (SDLC), from initial requirements gathering and design through implementation, testing, deployment, and maintenance.

Scalability and Efficiency: Focus on creating software that can efficiently handle increasing amounts of work or data and can be scaled up as needed.

Maintainability: Developing software that is easy to understand, modify, and extend, which facilitates future updates and the addition of new features.

Collaboration and Teamwork: Often requires collaboration among various stakeholders, including developers, designers, testers, project managers, and clients, to ensure all requirements are met and the project stays on track.

User-centric Design: Involves understanding and addressing the needs and experiences of users, ensuring the software is intuitive, accessible, and provides a positive user experience.

Risk Management: Identifying and mitigating potential risks throughout the development process, including technical, financial, and operational risks.

Documentation and Standards: Emphasis on thorough documentation and adherence to industry standards and best practices, ensuring consistency, reproducibility, and compliance.

Innovation and Adaptability: Staying current with emerging technologies, methodologies, and tools, and being able to adapt processes and techniques to meet new challenges and opportunities in the software industry.

These characteristics collectively ensure that software engineering produces high-quality, reliable, and maintainable software solutions that meet user requirements and are delivered on time and within budget.

Q.3 How risk handling has done in spiral model?

Ans: In the spiral model, risk handling is a central focus throughout the software development process. The spiral model, introduced by Barry Boehm in 1986, is an iterative and incremental approach that combines elements of both waterfall and prototyping models, emphasizing risk analysis and management. Here's how risk handling is addressed in the spiral model:

Iterative Cycles: The development process is divided into a series of iterative cycles, or spirals. Each cycle involves a sequence of steps that include planning, risk analysis, engineering, and evaluation. This allows for continuous assessment and mitigation of risks as the project progresses.

Risk Analysis Phase: In each cycle, a dedicated risk analysis phase is conducted. During this phase:

Identify Risks: Potential risks are identified based on current information about the project. These can include technical risks, financial risks, schedule risks, and user-related risks.

Assess Risks: Each identified risk is assessed in terms of its impact and likelihood. This helps prioritize the risks that need the most attention.

Develop Risk Mitigation Strategies: For each significant risk, strategies are developed to mitigate or manage the risk. This can include prototyping, simulations, alternative designs, or acquiring additional expertise.

Prototyping and Simulation: To mitigate technical risks, the spiral model often involves building prototypes or conducting simulations. These activities help validate requirements and design choices early, reducing the likelihood of costly errors later in the development process.

Stakeholder Involvement: Regular interaction with stakeholders is a key component of the spiral model. Stakeholders review the progress and provide feedback at the end of each cycle, which helps identify new risks and adjust strategies accordingly.

Review and Evaluation: At the end of each cycle, a thorough review and evaluation are conducted. This includes assessing the effectiveness of risk mitigation strategies and making

necessary adjustments for the next cycle. Decisions are made on whether to proceed with the next cycle, make changes, or even terminate the project if risks are deemed unmanageable.

Documentation and Communication: Detailed documentation of risks, mitigation strategies, and outcomes is maintained throughout the project. Clear communication channels are established to ensure that all team members and stakeholders are aware of the risks and how they are being addressed.

By incorporating these steps into each cycle of the development process, the spiral model ensures that risk handling is a continuous, proactive effort. This iterative approach allows for early identification and resolution of risks, leading to more robust and reliable software development outcomes.

Q.4 What are the various components of software model? **(AKTU 2021-22)**

Ans: A software model comprises several components that together define the structure, behavior, and development processes of a software system. These components are designed to ensure the software meets its requirements, is maintainable, and can be developed efficiently. The key components of a software model typically include:

Requirements Specification:

Functional Requirements: Detailed descriptions of the system's functionality and services.

Non-functional Requirements: Constraints on the services or functions, such as performance, security, and usability.

Architecture Design:

High-level Structure: The overall organization of the system, including the main components and their interactions.

Architectural Patterns: Common solutions to recurring design problems, such as MVC (Model-View-Controller), layered architecture, or microservices.

Component Design:

Detailed Design: Specifications for each component or module, including their interfaces, data structures, and algorithms.

Component Interaction: How components communicate and collaborate to achieve the desired functionality.

Data Model:

Data Structures: The organization of data within the system, such as tables, records, and files.

Database Design: The schema of the database, including tables, relationships, and constraints.

User Interface Design:

Layout and Navigation: The arrangement of visual elements and how users navigate through the system.

Interaction Design: How users interact with the system, including inputs, outputs, and feedback mechanisms.

Process Model:

Development Process: The methodology used to manage and execute the software development lifecycle (SDLC), such as Agile, Waterfall, or Spiral.

Project Management: Planning, scheduling, and tracking tasks, resources, and milestones.

Implementation:

Coding Standards: Guidelines for writing code to ensure consistency, readability, and maintainability.

Programming Languages: The languages and tools used to develop the software.

Testing and Validation:

Test Plans: Strategies and schedules for testing activities.

Test Cases: Specific conditions and inputs used to test the software's functionality and performance.

Validation: Ensuring the software meets user needs and requirements.

Deployment:

Deployment Plan: Steps and procedures for delivering the software to the end-users or production environment.

Environment Setup: Configuration of hardware, software, and network environments required for the software to run.

Maintenance and Support:

Bug Tracking: Systems for reporting, tracking, and resolving defects.

Updates and Patches: Procedures for releasing and applying software updates.

Documentation:

User Manuals: Guides and instructions for end-users.

Technical Documentation: Detailed information for developers and maintainers, including design documents, code comments, and API documentation.

Quality Assurance:

Quality Metrics: Measures used to assess the software's quality, such as defect density, code coverage, and performance benchmarks.

Review Processes: Peer reviews, code inspections, and walkthroughs to ensure quality and adherence to standards.

Each of these components plays a critical role in ensuring the successful development, deployment, and maintenance of software, providing a comprehensive framework for managing the complexities of software engineering.

Q.5 Iterative waterfall and spiral model for software lifecycle clean and discuss various activities in each phase. (AKTU 2022-23)

Ans: Both the Iterative Waterfall model and the Spiral model are used in software development lifecycles to address the complexities and risks inherent in creating software systems. Here's a detailed look at the phases and activities in each model:

Iterative Waterfall Model

The Iterative Waterfall model adapts the traditional Waterfall model by allowing feedback and iterations between phases. Here's an overview of the phases and activities:

Requirements Analysis and Specification:

Gather Requirements: Collect user and stakeholder requirements.

Analyze Requirements: Determine the feasibility and define the scope.

Document Requirements: Create detailed requirement specifications.

Review and Validate: Confirm requirements with stakeholders.

System and Software Design:

High-Level Design: Define the system architecture and high-level components.

Detailed Design: Specify the detailed workings of each component.

Review Designs: Conduct design reviews and iterate if necessary.

Implementation and Unit Testing:

Code Development: Write the code according to the design specifications.

Unit Testing: Test individual components for functionality.

Fix Issues: Resolve any defects found during unit testing.

Integration and System Testing:

Integrate Components: Combine individual components into a complete system.

System Testing: Test the integrated system to ensure it meets requirements.

Iterate as Needed: Return to previous phases if defects are found.

Deployment and Maintenance:

Deploy System: Release the system to the production environment.

User Training: Train users on the new system.

Maintenance: Address any issues, make necessary updates, and iterate if significant changes are needed.

Spiral Model

The Spiral model emphasizes risk management and iterative development, with each iteration called a "spiral." Each spiral involves several key activities:

Planning Phase:

Identify Objectives: Define the goals for the current iteration.

Determine Constraints: Identify constraints and conditions for development.

Plan Next Iteration: Develop a detailed plan for the activities in the next iteration.

Risk Analysis Phase:

Identify Risks: Identify potential risks for the current iteration.

Assess Risks: Evaluate the impact and likelihood of each risk.

Develop Risk Mitigation: Formulate strategies to mitigate identified risks, such as prototyping or simulations.

Review Risks: Conduct reviews with stakeholders and adjust plans as needed.

Engineering Phase:

Design: Create or refine the design based on the requirements and risk analysis.

Development: Implement the design by writing code.

Verification: Test the implementation to ensure it meets the specified requirements.

Documentation: Document the design, code, and testing results.

Evaluation Phase:

Stakeholder Review: Present the results of the current iteration to stakeholders for feedback.

Assess Progress: Evaluate whether the objectives have been met and identify any changes needed.

Prepare for Next Spiral: Plan for the next iteration, incorporating feedback and lessons learned.

In the Spiral model, these phases are repeated in each iteration, with the project gradually evolving through multiple spirals until the final system is completed and all risks are addressed. This approach ensures that risks are continuously managed and the system is incrementally refined based on feedback and changing requirements.

Both models aim to improve the traditional Waterfall approach by introducing flexibility and risk management, but they differ in their emphasis and specific processes. The Iterative Waterfall model allows for revisiting previous phases within a structured framework, while the Spiral model places a stronger focus on iterative risk assessment and stakeholder involvement.

Q.6 What are the reasons of software crisis? **(AKTU 2021-22)**

Ans: The "software crisis" refers to the difficulties and challenges faced in software development, particularly prevalent from the 1960s to the 1980s, but many aspects remain relevant today. The term was coined to describe the inability to develop software on time, within budget, and to meet requirements reliably. Here are the main reasons for the software crisis:

Increasing Complexity:

Large-Scale Systems: As software systems grew larger and more complex, managing their development became significantly more challenging.

Integration Challenges: Combining various system components, especially when they were developed independently, introduced unforeseen complexities.

Changing Requirements:

Evolving Needs: Requirements often change during the development process, but traditional methodologies struggled to accommodate these changes.

Unclear Requirements: Initial requirements were often incomplete or poorly understood, leading to frequent revisions and scope creep.

Lack of Proper Methodologies:

Ad-Hoc Development: Early software development was often unstructured, with many projects lacking a systematic approach.

Insufficient Planning: Many projects proceeded with inadequate planning, leading to unforeseen issues and delays.

Poor Project Management:

Inaccurate Estimations: Estimating the time, cost, and resources required for software development was notoriously difficult, often leading to overruns.

Inadequate Risk Management: Many projects did not effectively identify and manage risks, resulting in unexpected problems.

Inadequate Tools and Techniques:

Limited Automation: Early software development relied heavily on manual processes, which were time-consuming and error-prone.

Primitive Tools: The tools available for software design; development, testing, and maintenance were often rudimentary and insufficient for large projects.

Human Factors:

Skill Gaps: There was often a mismatch between the skills of the available workforce and the needs of the projects.

Communication Issues: Poor communication among team members, stakeholders, and users led to misunderstandings and errors.

Quality Assurance Challenges:

Insufficient Testing: Testing was often inadequate, leading to software that was buggy and unreliable.

Maintenance Difficulties: Once deployed, maintaining and updating software was difficult, especially when documentation was poor or absent.

Scalability Issues:

Performance Problems: Software that worked well on a small scale often failed to perform adequately when scaled up.

Resource Limitations: Hardware and infrastructure limitations sometimes constrained the effectiveness of software solutions.

Economic Pressures:

Budget Constraints: Limited financial resources led to cutting corners in development, testing, and quality assurance.

Market Pressures: The pressure to release products quickly to gain a competitive edge often resulted in rushed and incomplete software.

Legacy Systems:

Compatibility Issues: Integrating new software with existing legacy systems often proved difficult and costly.

Obsolescence: As technology rapidly evolved, older systems became obsolete, but replacing them was complex and expensive.

Addressing the software crisis led to the development of more rigorous software engineering practices, methodologies like Agile and DevOps, improved project management techniques, and advanced tools and technologies. These advancements have significantly mitigated many of the issues that characterized the software crisis, although challenges in software development still persist.

Q.7 Differentiate between software engineering process & conventional engineering process.

(AKTU 2018-19)

Ans: While both software engineering and conventional engineering processes aim to design, develop, and maintain systems that solve specific problems, they differ significantly in several aspects due to the inherent nature of software and traditional physical systems. Here are the key differences between the two:

Nature of the Product

Tangibility:

Software Engineering: Deals with intangible, logical products that consist of code and data. Software is essentially information and can be replicated perfectly with no material cost.

Conventional Engineering: Involves tangible, physical products such as buildings, bridges, machines, and electronics, which require materials and physical construction.

Complexity and Change:

Software Engineering: Software can be highly complex and is prone to frequent changes due to

evolving requirements, bug fixes, and updates.

Conventional Engineering: Physical systems are also complex but changes are usually less frequent and more costly once the system is built.

Process and Lifecycle

Development Process:

Software Engineering: Often follows iterative and incremental models like Agile, Scrum, or the Spiral model to handle changes and manage risks.

Conventional Engineering: Typically follows a more linear and sequential process, similar to the traditional Waterfall model, with distinct phases for design, construction, and testing.

Prototyping and Testing:

Software Engineering: Prototyping is common, and testing is integrated throughout the development lifecycle, allowing for ongoing refinement and immediate feedback.

Conventional Engineering: Prototyping is used but often at a smaller scale or in simulation. Full-scale testing usually occurs after the design and construction phases.

Design and Documentation

Design Flexibility:

Software Engineering: Design can be more flexible, allowing for changes and iterations even late in the development process without significant physical constraints.

Conventional Engineering: Design changes are more constrained by physical laws and material properties. Late changes can be costly and time-consuming.

Documentation:

Software Engineering: Requires detailed documentation for requirements, design, code, testing, and maintenance. Documentation is crucial for understanding and modifying the software.

Conventional Engineering: Also requires comprehensive documentation, including blueprints, material specifications, and maintenance manuals. However, the documentation often remains more static once the project is complete.

Quality Assurance and Maintenance

Quality Assurance:

Software Engineering: Continuous integration, automated testing, and regular updates are key practices. Quality assurance is an ongoing process.

Conventional Engineering: Quality assurance is focused more on the construction phase and initial testing. Once built, physical systems undergo regular maintenance checks but are less frequently "updated."

Maintenance:

Software Engineering: Maintenance involves bug fixes, updates, enhancements, and adapting to new environments. It is a continuous and integral part of the software lifecycle.

Conventional Engineering: Maintenance focuses on repairs, routine servicing, and ensuring the longevity of physical components. Upgrades and modifications are less frequent.

Risk and Cost Management

Risk Management:

Software Engineering: Risk is managed through iterative development, continuous testing, and

stakeholder feedback. Risks can often be mitigated through software patches and updates.

Conventional Engineering: Risk management involves thorough upfront planning, safety assessments, and material testing. Physical risks are often more critical and harder to mitigate post-construction.

Cost:

Software Engineering: Initial development costs can be high, but replication and distribution costs are low. Maintenance and support are ongoing costs.

Conventional Engineering: High initial costs due to materials, labor, and construction. Maintenance and repair can also be expensive, particularly for large-scale systems.

Evolution and Innovation

Innovation Pace:

Software Engineering: Rapid pace of innovation with new tools, languages, frameworks, and methodologies constantly emerging.

Conventional Engineering: Innovation is steady but often slower due to the dependency on material science advancements and physical prototyping constraints.

Scalability and Distribution:

Software Engineering: Software can be easily scaled and distributed to millions of users with minimal additional cost.

Conventional Engineering: Scaling physical systems involves significant costs in materials and labor, and distribution is limited by physical and logistical constraints.

Summary

In summary, software engineering is characterized by its focus on managing complexity, accommodating change, and emphasizing iterative processes. Conventional engineering deals with physical systems where material constraints and linear processes play a more significant role. Both fields require meticulous planning, design, documentation, and quality assurance, but their methodologies and challenges differ due to the fundamental nature of the products they create.

Q.8 Discuss various software qualities attributes.

Ans: Software quality attributes, also known as non-functional requirements, define the characteristics and behavior of a software system that affect its performance, maintainability, usability, and other key factors. These attributes are crucial for ensuring that the software not only meets its functional requirements but also performs effectively and efficiently in various environments. Here are some of the primary software quality attributes:

1. Functionality

Correctness: The degree to which the software performs its intended functions accurately.

Interoperability: The ability of the software to interact and work with other systems or components.

Compliance: Adherence to relevant standards, laws, and regulations.

Security: Protection of data and resources from unauthorized access and vulnerabilities.

2. Reliability

Availability: The degree to which the software is operational and accessible when required.

Fault Tolerance: The ability of the software to continue operating correctly in the event of a failure.

Recoverability: The capability of the software to recover data and resume operations after a failure.

3. Usability

Learn ability: How easy it is for users to learn and become proficient with the software.

Operability: The ease with which users can operate and control the software.

User Interface Aesthetics: The visual appeal and consistency of the user interface.

Accessibility: Ensuring the software can be used by people with disabilities.

4. Efficiency

Performance: The speed and responsiveness of the software under various conditions.

Resource Utilization: The degree to which the software uses system resources like CPU, memory, and storage efficiently.

Scalability: The ability of the software to handle increasing loads or be expanded to accommodate growth.

5. Maintainability

Modularity: The degree to which the software is composed of discrete components that can be modified independently.

Reusability: The extent to which parts of the software can be used in other applications.

Analyzability: The ease with which the software can be analyzed for faults, performance issues, or other attributes.

Modifiability: The ease with which the software can be changed to correct faults, improve performance, or adapt to a changed environment.

Testability: The degree to which the software supports testing activities.

6. Portability

Adaptability: The ease with which the software can be adapted to different environments.

Install ability: The ease with which the software can be installed and set up.

Conformance: The degree to which the software adheres to standards that facilitate portability.

Replace ability: The ease with which the software can be replaced by another system.

7. Reusability

The ability of the software components to be reused in different applications or in different parts of the same application to reduce redundancy and effort.

8. Compatibility

Co-existence: The capability of the software to exist alongside other systems and applications without adverse effects.

Interoperability: The ability of the software to interact with other systems or components seamlessly.

9. Modifiability

Flexibility: The ease with which the software can accommodate changes in requirements or environments.

Extensibility: The ease with which new functionality can be added to the software.

10. Supportability

Serviceability: The ability of the software to be easily maintained and supported.

Configurability: The ability to configure the software to meet various user needs.

These software quality attributes collectively ensure that the software is not only functional but also reliable, efficient, user-friendly, and maintainable. They are critical for the long-term success and sustainability of software systems, impacting everything from user satisfaction to the cost of future development and maintenance.

Q.9 What is the need of SDLC? Discuss evolutionary development model in detail with the help of diagram. **(AKTU 2021-22)**

Ans: The Need for SDLC

The Software Development Life Cycle (SDLC) is essential for several reasons:

Structured Approach: It provides a systematic and organized way to develop software, ensuring that all aspects of the project are addressed.

Project Management: Helps in planning, scheduling, and tracking the progress of the project.

Quality Assurance: Ensures that the software meets quality standards and requirements through various stages of testing and validation.

Risk Management: Identifies and mitigates risks early in the development process.

Cost Management: Helps in estimating costs and managing the budget effectively.

Improved Communication: Enhances communication among stakeholders, developers, and users, ensuring that everyone has a clear understanding of the project goals and progress.

Maintenance and Support: Facilitates easier maintenance and updates to the software after deployment.

Evolutionary Development Model

The Evolutionary Development Model is an iterative approach that combines elements of both iterative and incremental models. It focuses on developing a system through repeated cycles (iterations) and in smaller portions (increments) at a time, allowing partial implementations of the system to evolve and improve based on user feedback and changing requirements.

Key Characteristics:

Incremental Delivery: The system is built and delivered in small, manageable increments, each providing additional functionality.

User Involvement: Users provide feedback after each iteration, which is used to refine requirements and improve the system.

Flexibility: The model is flexible and can accommodate changes in requirements at any stage of development.

Risk Reduction:

By developing the system incrementally, risks are identified and mitigated early in the process.

Phases of Evolutionary Development Model:

Initial Planning:

Define the overall scope and objectives of the project.

Identify high-level requirements and constraints.

Specification and Analysis:

Gather detailed requirements for the initial increment.
Analyze requirements to ensure they are feasible and clear.

Design:
Create a high-level design for the system.
Design the specific increment to be developed.

Implementation:
Develop the increment according to the design specifications.
Conduct unit testing to ensure the increment functions correctly.

Testing and Integration:
Perform system testing on the increment.
Integrate the increment with previously developed parts of the system.

Evaluation and Feedback:
Deliver the increment to users for feedback.
Gather feedback and analyze it to identify improvements and changes.

Iteration Planning:
Plan the next iteration based on feedback and updated requirements.
Repeat the cycle for the next increment.

Q.10 Define generic software with example. **(AKTU 2022-23)**

Ans: Generic software refers to software applications or systems that are designed to be flexible and adaptable, capable of serving a wide range of users or purposes without significant customization or modification. These types of software are often characterized by their general-purpose nature, scalability, and ability to be configured to meet diverse needs.

Example: An example of generic software is a spreadsheet application like Microsoft Excel or Google Sheets. These applications are designed to provide users with a versatile platform for organizing, analyzing, and presenting data in various formats. While they come with pre-built features and functions for common tasks such as calculations, data visualization, and formulae, users can customize and extend their functionality to suit specific requirements.

Key features of generic software like spreadsheet applications include:

Versatility: They can be used for a wide range of tasks across different industries and domains, such as financial analysis, project management, inventory tracking, and academic research.

Customization: Users can customize the software by creating custom formulas, macros, templates, and formatting options to tailor it to their specific needs and preferences.

Scalability: They can handle large datasets and complex calculations, making them suitable for both individual users and organizations of all sizes.

Ease of Use: Generic software often comes with intuitive user interfaces and built-in help resources, making them accessible to users with varying levels of technical expertise.

Interoperability: They can integrate with other software applications and systems, allowing users to import and export data, collaborate with others, and automate workflows.

Overall, generic software provides users with a flexible and adaptable platform that can be used for a wide range of tasks, making it a valuable tool for individuals, businesses, and organizations across various industries.

Q.11 Explain Software Quality Attributes in detail. **(AKTU 2023-24)**

Software quality attributes are key characteristics or criteria used to assess the overall quality of a software system. They represent the non-functional requirements (NFRs) that determine how well a software system performs under various conditions. Quality attributes guide decisions during the software design, development, testing, and maintenance phases, as they address issues beyond just the functional correctness of the software. Here's a detailed explanation of the most common and important software quality attributes:

1. Performance

Performance refers to how efficiently the software system utilizes resources (e.g., CPU, memory, bandwidth) and responds to inputs within acceptable time limits.

- **Response Time**: The time it takes to process a request or action.
- **Throughput**: The number of transactions processed in a given period.
- **Latency**: The delay before a transfer of data begins following an instruction for its transfer.

2. Scalability

Scalability is the ability of a system to handle growth, whether in terms of the number of users, the amount of data, or the complexity of operations, without compromising performance.

- **Vertical Scalability (Scaling up)**: Adding more resources (e.g., CPU, memory) to a single machine.
- **Horizontal Scalability (Scaling out)**: Distributing the load across multiple machines or servers.

3. Reliability

Reliability refers to the software's ability to perform its intended functions consistently without failure over time.

- **Mean Time Between Failures (MTBF)**: The average time between system failures.
- **Fault Tolerance**: The ability of the system to recover from failures and continue functioning.

4. Availability

Availability is the proportion of time a system is operational and accessible to users. It is often expressed as a percentage (e.g., 99.9% uptime).

  - **Downtime**: The time the system is unavailable.
  - **Redundancy**: Duplicate systems or components that can take over in case of failure.

5. Maintainability

Maintainability refers to the ease with which a software system can be updated, fixed, or improved over time.

- **Modularity**: The degree to which the system is broken down into smaller, manageable components.
- **Code Readability**: The clarity and simplicity of the code, which impacts how easily it can be understood and modified.
- **Testability**: The ease of testing the system for correctness and performance.

6. Usability

Usability is the ease with which users can interact with the software to achieve their goals effectively, efficiently, and satisfactorily.

- **User Interface Design**: How intuitive and user-friendly the interface is.
- **User Experience (UX)**: The overall experience of the user when interacting with the system.

7. Security

Security refers to the ability of the software to protect against unauthorized access, data breaches, and other security threats.

- **Authentication**: Ensuring that users are who they say they are.
- **Authorization**: Ensuring users can only access resources they are permitted to.
- **Data Encryption**: Protecting sensitive data during storage and transmission.

8. Portability

Portability is the ease with which software can be transferred from one environment or platform to another.

- **Platform Independence**: The ability of the software to run on different operating systems or devices with minimal modification.
- **Configuration Flexibility**: How easily the software can be configured for different environments..

9. Interoperability

Interoperability refers to the ability of a software system to work seamlessly with other systems, applications, or platforms.

- **Data Exchange**: The ability to share data with other systems in compatible formats.
- **Standard Compliance**: Adherence to standards (e.g., API, file formats) that ensure compatibility.

10. Flexibility

Flexibility is the ability of a software system to adapt to changing requirements or conditions with minimal effort.

- **Configurability**: The extent to which the system can be adjusted to meet new needs without changing the code.
- **Extensibility**: The ease with which new features or components can be added.

Q.12 Illustrate the statement "Software engineering is layered technology". **(AKTU 2023-24)**

The statement "Software engineering is layered technology" can be illustrated as a series of layers stacked on top of each other, where each layer represents a different level of abstraction or different set of technologies that work together to create software.

Here's a conceptual illustration:

Layered Diagram of Software Engineering

1. **Hardware Layer** (Bottom layer)

- This is the physical infrastructure on which software runs. It includes computers, servers, and other hardware components.

2. **Operating System Layer**

- The operating system (OS) manages hardware resources and provides a platform for running software. It includes file systems, memory management, and device drivers.

3. **Middleware Layer**

- Middleware facilitates communication between different software components. Examples include databases, network protocols, and message queues that allow applications to interact with each other.

4. **Application Framework Layer**

- This layer includes predefined structures or libraries that developers use to build specific applications (e.g., web frameworks like Django, React for front-end, or backend frameworks like Spring).

5. **Software Application Layer**

- The specific software application or service being developed. It represents the end-user product (e.g., a web app, mobile app, or enterprise software) built using tools, libraries, and frameworks.

6. **User Interface Layer** (Topmost layer)

- This is the layer that users interact with directly. It includes graphical elements like buttons, menus, and forms through which users can interact with the application.

BUDDHA SERIES

(Unit Wise Solved Question & Answers)

Course – B.Tech. (CSE)

College – Buddha Institute of Technology

(AKTU CODE-525)

Department: Computer Science & Engineering (AIML/DS)

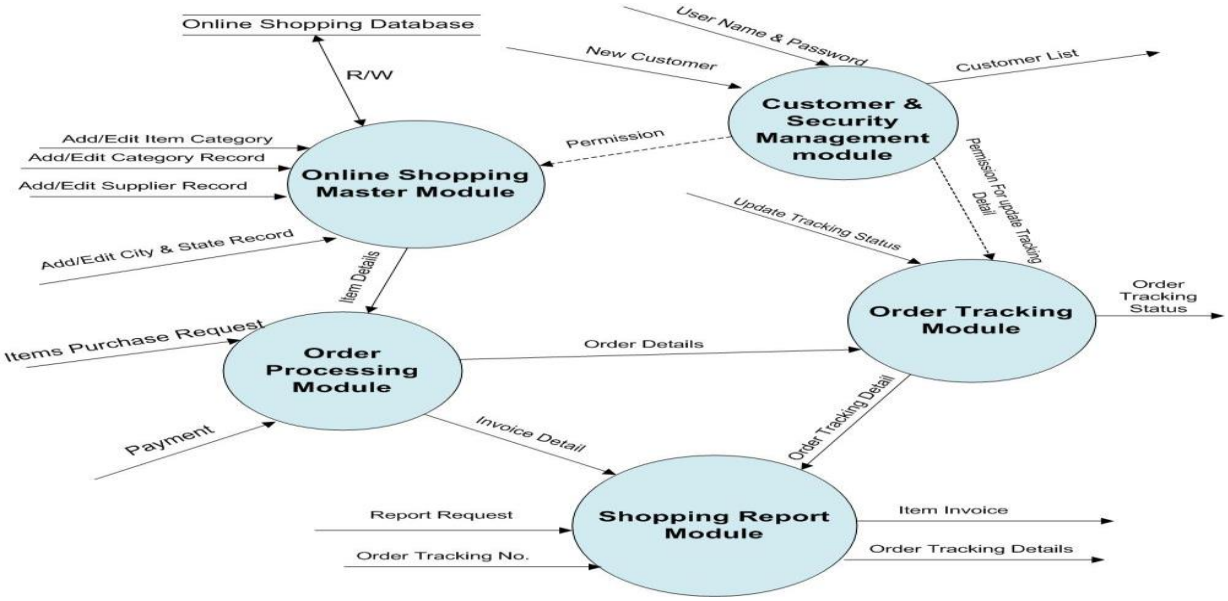
Subject: Software Engineering (BCS 601)

Faculty Name: Mr. Satish Kumar

Unit - 2

Q.1- Draw zero and one level data flow diagram for railway reservation system. **(AKTU 2018-19)**

Ans:



Q.2 What are the various stages of requirement engineering process? Explain it with diagrammatic representation. **(AKTU 2022-23)**

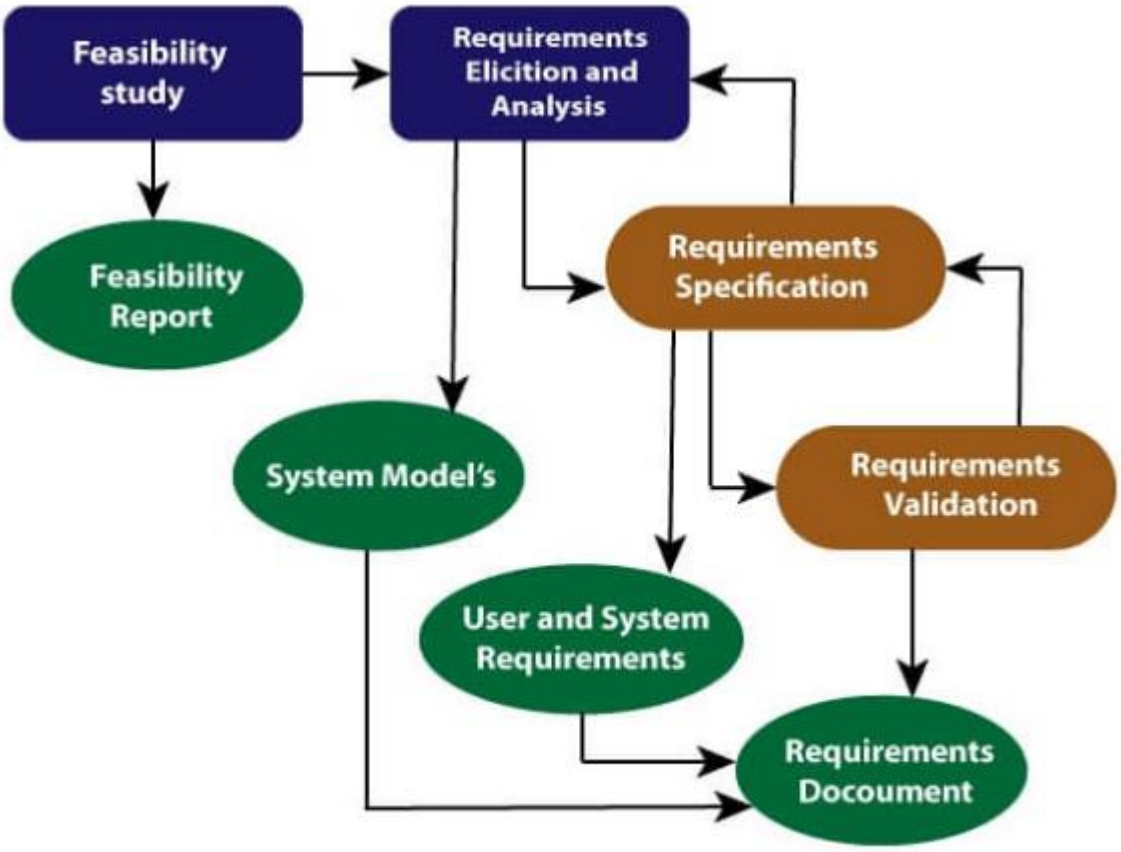
Ans: Requirements engineering (RE) refers to the process of defining, documenting, and maintaining requirements in the engineering design process. Requirement engineering provides the appropriate mechanism to understand what the customer desires, analyzing the need, and assessing feasibility, negotiating a reasonable solution, specifying the solution clearly, validating the specifications and managing the requirements as they are transformed into a working system. Thus, requirement engineering is the disciplined application of proven principles, methods, tools, and notation to describe a proposed system's intended behavior and its associated constraints.

Requirement Engineering Process

It is a four-step process, which includes -

1. Feasibility Study

2. Requirement Elicitation and Analysis
3. Software Requirement Specification
4. Software Requirement Validation
5. Software Requirement Management



Requirement Engineering Process

1. Feasibility Study:

The objective behind the feasibility study is to create the reasons for developing the software that is acceptable to users, flexible to change and conformable to established standards.

Types of Feasibility:

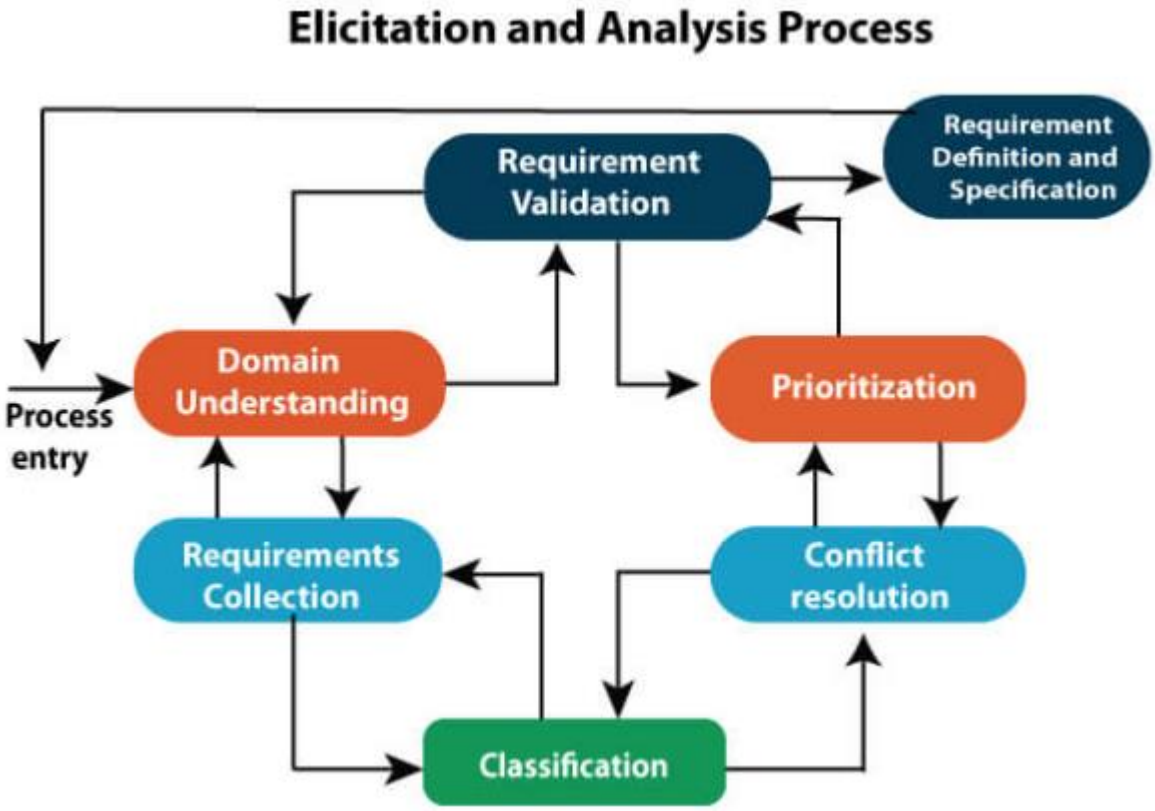
1. **Technical Feasibility** - Technical feasibility evaluates the current technologies, which are needed to accomplish customer requirements within the time and budget.
2. **Operational Feasibility** - Operational feasibility assesses the range in which the required software performs a series of levels to solve business problems and customer requirements.

3. **Economic Feasibility** - Economic feasibility decides whether the necessary software can generate financial profits for an organization.

2. Requirement Elicitation and Analysis:

This is also known as the gathering of requirements. Here, requirements are identified with the help of customers and existing systems processes, if available.

Analysis of requirements starts with requirement elicitation. The requirements are analyzed to identify inconsistencies, defects, omission, etc. We describe requirements in terms of relationships and also resolve conflicts if any.



Problems of Elicitation and Analysis

- Getting all, and only, the right people involved.
- Stakeholders often don't know what they want
- Stakeholders express requirements in their terms.
- Stakeholders may have conflicting requirements.
- Requirement change during the analysis process.
- Organizational and political factors may influence system requirements.

3. Software Requirement Specification:

Software requirement specification is a kind of document which is created by a software analyst after the requirements collected from the various sources - the requirement received by the customer written in ordinary language. It is the job of the analyst to write the requirement in technical language so that they can be understood and beneficial by the development team.
The models used at this stage include ER diagrams, data flow diagrams (DFDs), function decomposition diagrams (FDDs), data dictionaries, etc.

- **Data Flow Diagrams:** Data Flow Diagrams (DFDs) are used widely for modeling the requirements. DFD shows the flow of data through a system. The system may be a company, an organization, a set of procedures, a computer hardware system, a software system, or any combination of the preceding. The DFD is also known as a data flow graph or bubble chart.
- **Data Dictionaries:** Data Dictionaries are simply repositories to store information about all data items defined in DFDs. At the requirements stage, the data dictionary should at least define customer data items, to ensure that the customer and developers use the same definition and terminologies.
- **Entity-Relationship Diagrams:** Another tool for requirement specification is the entity-relationship diagram, often called an "E-R diagram." It is a detailed logical representation of the data for the organization and uses three main constructs i.e. data entities, relationships, and their associated attributes.

4. Software Requirement Validation:

After requirement specifications developed, the requirements discussed in this document are validated. The user might demand illegal, impossible solution or experts may misinterpret the needs. Requirements can be the check against the following conditions -

- If they can practically implement
- If they are correct and as per the functionality and specially of software
- If there are any ambiguities
- If they are full
- If they can describe

Requirements Validation Techniques

- **Requirements reviews/inspections:** systematic manual analysis of the requirements.
- **Prototyping:** Using an executable model of the system to check requirements.
- **Test-case generation:** Developing tests for requirements to check testability.

- **Automated consistency analysis:** checking for the consistency of structured requirements descriptions.

Software Requirement Management:

Requirement management is the process of managing changing requirements during the requirements engineering process and system development.

New requirements emerge during the process as business needs a change, and a better understanding of the system is developed.

The priority of requirements from different viewpoints changes during development process.

The business and technical environment of the system changes during the development.

Prerequisite of Software requirements

Collection of software requirements is the basis of the entire software development project. Hence they should be clear, correct, and well-defined.

A complete Software Requirement Specifications should be:

- Clear
- Correct
- Consistent
- Coherent
- Comprehensible
- Modifiable
- Verifiable
- Prioritized
- Unambiguous
- Traceable
- Credible source

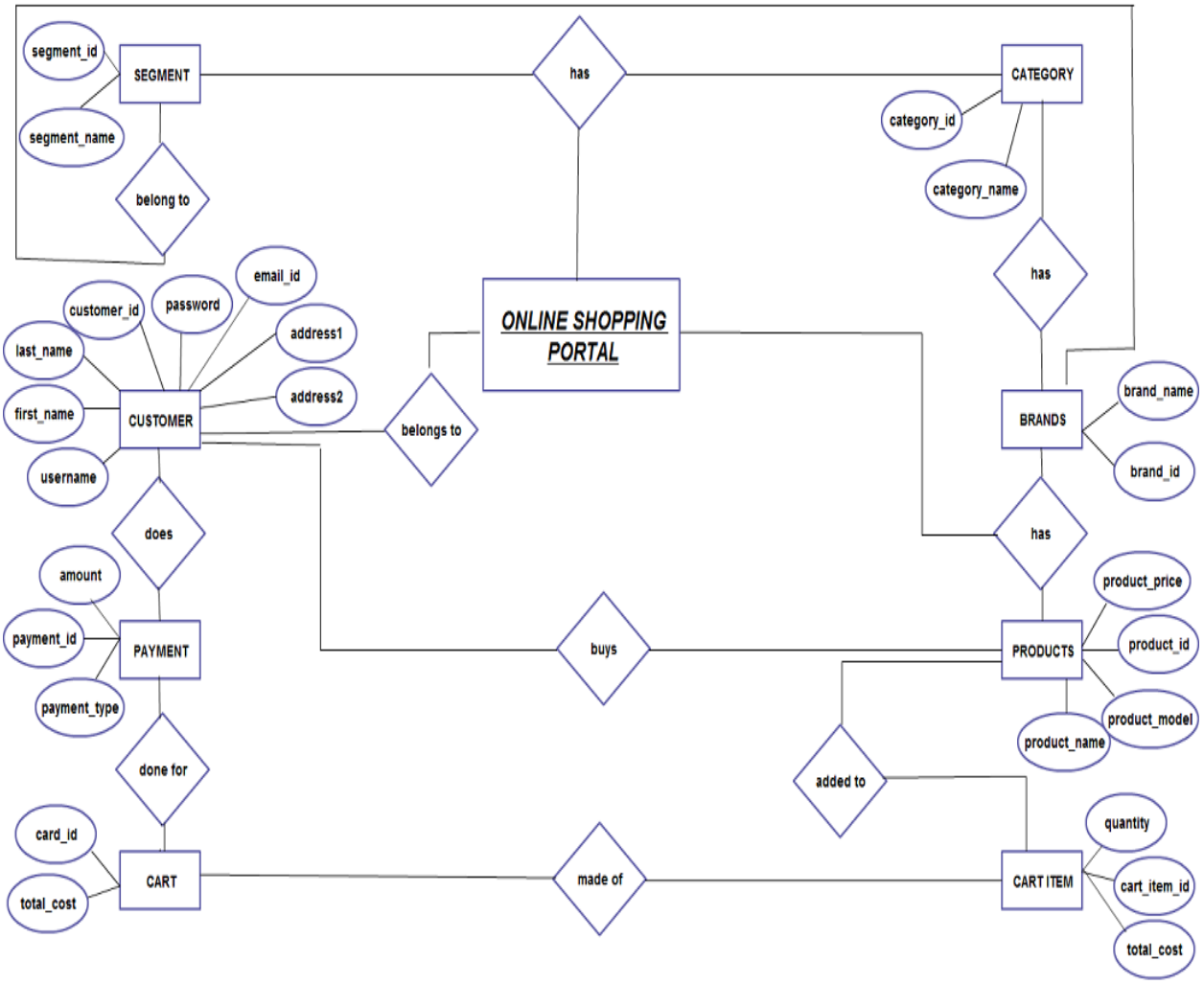
Software Requirements: Largely software requirements must be categorized into two categories:

1. **Functional Requirements:** Functional requirements define a function that a system or system element must be qualified to perform and must be documented in different forms. The functional requirements are describing the behavior of the system as it correlates to the system's functionality.
2. **Non-functional Requirements:** This can be the necessities that specify the criteria that can be used to decide the operation instead of specific behaviors of the system. Non-functional requirements are divided into two main categories:
    - **Execution qualities** like security and usability, which are observable at run time.

- **Evolution qualities** like testability, maintainability, extensibility, and scalability that embodied in the static structure of the software system.

Q.3 Draw ER-diagram for online shopping system**.** **(AKTU 2017-18)**

Ans:



Q.4 Explain ISO 9000 series. Write down the procedures for getting an ISO certificate.

(AKTU 2018-19)

Ans: ISO 9000 Certification

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

Types of ISO 9000 Quality Standards

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

1. **ISO 9001:** This standard applies to the organizations engaged in design, development, production, and servicing of goods. This is the standard that applies to most software development organizations.
2. **ISO 9002:** This standard applies to those organizations which do not design products but are only involved in the production. Examples of these category industries contain steel and car manufacturing industries that buy the product and plants designs from external sources and are engaged in only manufacturing those products. Therefore, ISO 9002 does not apply to software development organizations.
3. **ISO 9003:** This standard applies to organizations that are involved only in the installation and testing of the products. For example, Gas companies.

How to get ISO 9000 Certification?

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

1. **Application:** Once an organization decided to go for ISO certification, it applies to the registrar for registration.
2. **Pre-Assessment:** During this stage, the registrar makes a rough assessment of the organization.
3. **Document review and Adequacy of Audit:** During this stage, the registrar reviews the document submitted by the organization and suggest an improvement.
4. **Compliance Audit:** During this stage, the registrar checks whether the organization has compiled the suggestion made by it during the review or not.

5. **Registration:** The Registrar awards the ISO certification after the successful completion of all the phases.
6. **Continued Inspection:** The registrar continued to monitor the organization time by time.

Q.5 What is known as SRS review? How is it conducted? **(AKTU 2018-19)**

Ans: SRS stands for Software Requirements Specification, which is a comprehensive document that outlines the functional and non-functional requirements of a software system. An SRS review, also known as a Software Requirements Specification review, is a formal process of examining and evaluating the SRS document to ensure its completeness, accuracy, clarity, and consistency.

Purpose of SRS Review:

Validation: Ensure that the requirements accurately reflect the needs and expectations of stakeholders.

Verification: Confirm that the requirements are clear, complete, consistent, and feasible.

Communication: Enhance communication among project stakeholders by ensuring a common understanding of the requirements.

Risk Identification: Identify potential issues or risks early in the development process to mitigate them effectively.

Baseline: Establish a baseline for the software development process, providing a reference point for future activities.

Conducting an SRS Review:

Preparation:

Schedule the review meeting and invite relevant stakeholders, including project managers, developers, testers, and end-users.

Distribute copies of the SRS document to all participants well in advance of the review meeting to allow for thorough preparation.

Review Meeting:

Conduct the review meeting in a structured manner, led by a facilitator who guides the discussion and ensures all aspects of the SRS document are covered.

Use a checklist or predefined criteria to evaluate the document systematically, focusing on aspects such as completeness, correctness, clarity, consistency, and feasibility.

Encourage active participation from all stakeholders, allowing them to provide feedback, raise concerns, and ask questions about the requirements.

Document all comments, suggestions, and decisions made during the review meeting for future reference.

Resolution of Issues:

After the review meeting, compile a list of identified issues, discrepancies, or areas for improvement in the SRS document.

Assign responsibility for addressing each issue to specific individuals or teams, ensuring accountability for follow-up actions.

Prioritize the identified issues based on their severity, impact, and urgency, focusing on resolving

critical issues first.

Revision and Approval:

Incorporate the necessary revisions and updates into the SRS document based on the feedback received during the review.

Circulate the revised SRS document to all stakeholders for final review and approval.

Obtain sign-off from key stakeholders to indicate their acceptance of the updated requirements.

Documentation:

Document the outcomes of the SRS review, including any changes made to the document, decisions taken, and actions assigned.

Maintain a record of the review process for future reference and audit purposes.

Best Practices for SRS Review:

Prepare in Advance: Reviewers should thoroughly read the SRS document before the meeting to identify potential issues.

Focus on Collaboration: Encourage open communication and collaboration among stakeholders to ensure a comprehensive review.

Follow-Up: Ensure that all identified issues are addressed promptly and that the updated SRS document reflects the agreed-upon changes.

Iterative Process: Recognize that SRS review may require multiple iterations to achieve consensus and finalize the requirements.

By conducting thorough SRS reviews, organizations can improve the quality of their software requirements, reduce the likelihood of misunderstandings and errors, and ultimately increase the success rate of software development projects.

Q.6 Write benefits of SQA**.** **(AKTU 2017-18)**

Ans: Software Quality Assurance (SQA) is a set of systematic activities designed to ensure that software products and processes conform to specified requirements, standards, and procedures. SQA plays a crucial role in achieving and maintaining high-quality software throughout its lifecycle. Here are some of the key benefits of implementing SQA practices:

Improved Product Quality: SQA activities help identify defects, errors, and inconsistencies early in the development process, leading to higher-quality software products that meet user needs and expectations.

Reduced Costs: By detecting and addressing issues early, SQA helps prevent costly rework, delays, and customer dissatisfaction. This results in lower development and maintenance costs over the long term.

Enhanced Customer Satisfaction: High-quality software products that are reliable, usable, and meet user requirements contribute to increased customer satisfaction and loyalty.

Early Risk Identification and Mitigation: SQA practices help identify potential risks and uncertainties in software projects early, allowing teams to proactively address them before they escalate into major problems.

Compliance with Standards and Regulations: SQA ensures that software products and processes adhere to industry standards, regulations, and best practices, reducing legal and

regulatory risks for organizations.

Increased Productivity and Efficiency: SQA practices such as process automation, documentation, and training contribute to improved productivity and efficiency by streamlining development workflows and reducing manual effort.

Continuous Process Improvement: SQA promotes a culture of continuous improvement by providing feedback, metrics, and insights into software development processes, enabling teams to identify areas for enhancement and optimization.

Facilitated Decision-Making: SQA provides stakeholders with accurate and timely information about the quality and status of software projects, enabling informed decision-making and risk management.

Enhanced Collaboration and Communication: SQA encourages collaboration and communication among cross-functional teams, fostering a shared understanding of quality goals and responsibilities.

Increased Competitiveness: High-quality software products give organizations a competitive edge in the marketplace, attracting customers and opportunities for business growth.

Establishment of Best Practices: SQA helps establish and institutionalize best practices, methodologies, and quality standards within an organization, ensuring consistency and repeatability in software development processes.

Improved Reputation and Brand Image: Consistently delivering high-quality software products enhances an organization's reputation and brand image, leading to increased trust and credibility among customers, partners, and stakeholders.

Overall, SQA is essential for ensuring the success, sustainability, and competitiveness of software products and organizations in today's dynamic and competitive business environment.

Q.7 Construct decision table for largest number among three numbers**.** **(AKTU 2017-18)**

Ans: A decision table is a tabular representation of decision logic, often used to illustrate complex decision-making processes. It consists of conditions, actions, and rules that map conditions to actions. Here, we'll construct a decision table for determining the largest number among three given numbers A, B & C.

1. **Conditions:**
   - $A \geq B$
   - $A \geq C$
   - $B \geq C$
2. **Actions:**
   - Print A (indicating A is the largest)
   - Print B (indicating B is the largest)
   - Print C (indicating C is the largest)

Rule	Condition 1: $A \geq B$	Condition 2: $A \geq C$	Condition 3: $B \geq C$	Action 1: Print A	Action 2: Print B	Action 3: Print C
1	T	T	-	X		
2	T	F	T		X	
3	T	F	F			X
4	F	T	-			X
5	F	F	T		X	
6	F	F	F			X

Explanation of Rules

1. **Rule 1**: If $A \geq B$ and $A \geq C$, then A is the largest.
2. **Rule 2**: If $A \geq B$ but $A < C$ and $B \geq C$, then B is the largest.
3. **Rule 3**: If $A \geq B$ but $A < C$ and $B < C$, then C is the largest.
4. **Rule 4**: If $A < B$ and $A \geq C$, B must be greater than both A and C. Thus, B is the largest.
5. **Rule 5**: If $A < B$ and $A < C$ and $B \geq C$, then B is the largest.
6. **Rule 6**: If $A < B$ and $A < C$ and $B < C$, then C is the largest.

Q.8 List the points of differences between Verification and Validation. **(AKTU 2021-22)**

Ans: **Verification-**

Verification is the process of checking that software achieves its goal without any bugs. It is the process to ensure whether the product that is developed is right or not. It verifies whether the developed product fulfills the requirements that we have. Verification is static testing. Verification means **Are we building the product right?**

Validation-

Validation is the process of checking whether the software product is up to the mark or in other words product has high-level requirements. It is the process of checking the validation of the product i.e. it checks what we are developing is the right product. It is validation of the actual and expected products. Validation is dynamic testing. Validation means **Are we building the right product?**

Verification	Validation
Are we implementing the system right?	Are we implementing the right system?
Evaluating products of a development phase	Evaluating products at the closing of the development process
The objective is making sure the product is as per the requirements and design specifications	The objective is making sure that the product meets user's requirements
Activities included: reviews, meetings, and inspections	Activities included: black box testing, white box testing, and grey box testing
Verifies that outputs are according to inputs or not	Validates that the users accept the software or not
Items evaluated: plans, requirement specifications, design specifications, code, and test cases	Items evaluated: actual product or software under test
Manual checking of the documents and files	Checking the developed products using the documents and files

Q.9 Discuss the importance of software specification Document. And also explain the typical IEEE format of SRS document. **(AKTU 2021-22)**

Ans: An SRS forms the basis of an organization's entire project. It sets out the framework that all the development teams will follow. It provides critical information to all the teams, including development, operations, quality assurance (QA) and maintenance, ensuring the teams are in agreement.
Depending upon information gathered after interaction, SRS is developed which describes requirements of software that may include changes and modifications that is needed to be done to increase quality of product and to satisfy customer's demand.

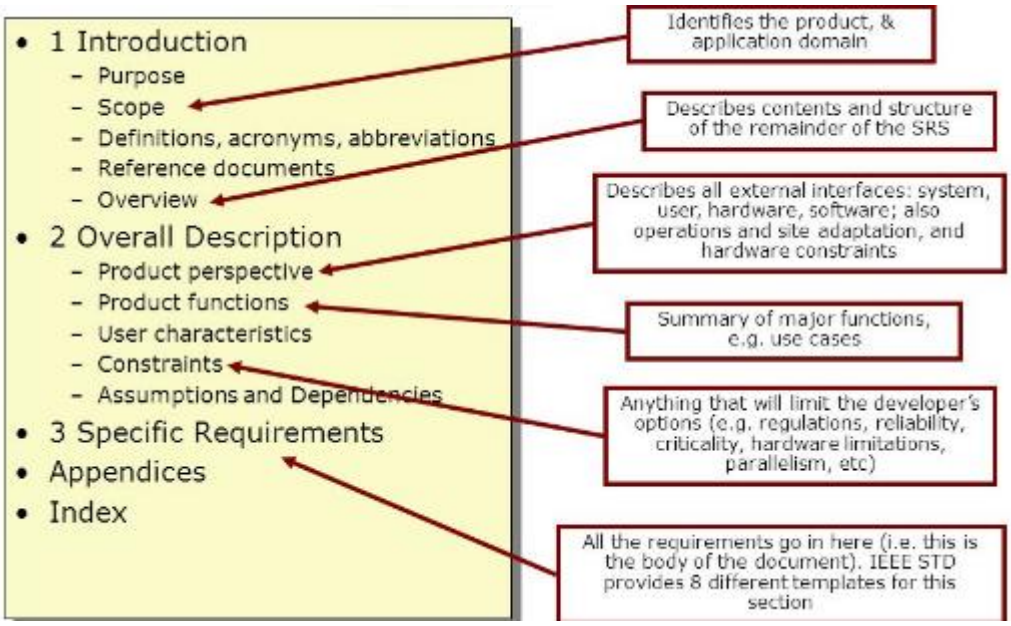
1. Introduction:
(i) Purpose of this Document – At first, main aim of why this document is necessary and what's purpose of document is explained and described.
(ii) Scope of this document – In this, overall working and main objective of document and what value it will provide to customer is described and explained. It also includes a description of development cost and time required.
(iii) Overview – In this, description of product is explained. It's simply summary or overall review of product.
2. General description : In this, general functions of product which includes objective of user, a user characteristic, features, benefits, about why its importance is mentioned. It also describes features of user community.
3. Functional Requirements: In this, possible outcome of software system which includes effects due to operation of program is fully explained. All functional requirements which may include calculations, data processing, etc. are placed in a ranked order.
4. Interface Requirements: In this, software interfaces which mean how software program communicates with each other or users either in form of any language, code, or message are fully described or explained. Examples can be shared memory, data streams, etc.
5. Performance Requirements: In this, how a software system performs desired functions under specific condition is explained. It also explains required time, required memory, maximum error rate, etc.

6. Design Constraints: In this, constraints which simply mean limitation or restriction are specified and explained for design team. Examples may include use of a particular algorithm, hardware and software limitations, etc.

7. Non-Functional Attributes: In this, non-functional attributes are explained that are required by software system for better performance. An example may include Security, Portability, Reliability, Reusability, Application compatibility, Data integrity, Scalability capacity, etc.

8. Preliminary Schedule and Budget : In this, initial version and budget of project plan are explained which include overall time duration required and overall cost required for development of project.

9. Appendices: In this, additional information like references from where information is gathered, definitions of some specific terms, acronyms, abbreviations, etc. are given and explained.



Q.10 Explain the SEI-CMM model. What do you mean by state of fire fighting? **(AKTU 2022-23)**

Ans: Software Engineering Institute Capability Maturity Model (SEICMM)

The Capability Maturity Model (CMM) is a procedure used to develop and refine an organization's software development process.

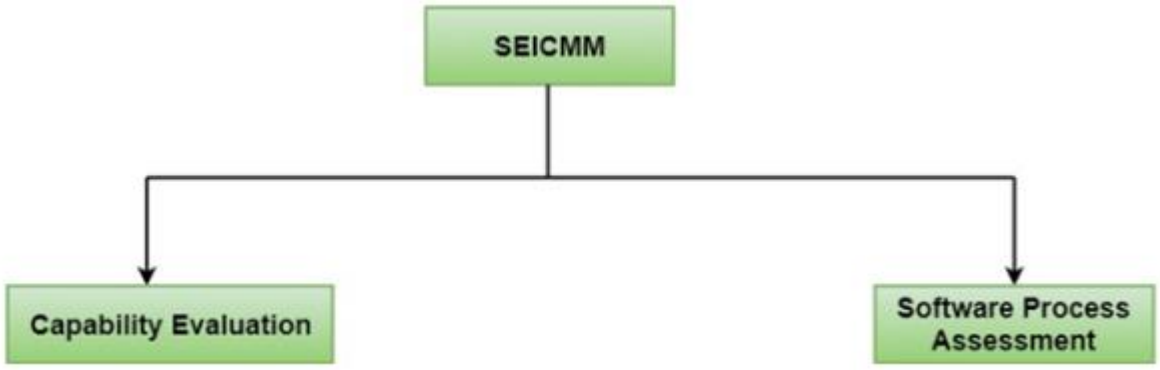
The model defines a five-level evolutionary stage of increasingly organized and consistently more mature processes.

CMM was developed and is promoted by the Software Engineering Institute (SEI), a research and development center promote by the U.S. Department of Defense (DOD).

Capability Maturity Model is used as a benchmark to measure the maturity of an organization's software process.

Methods of SEI-CMM

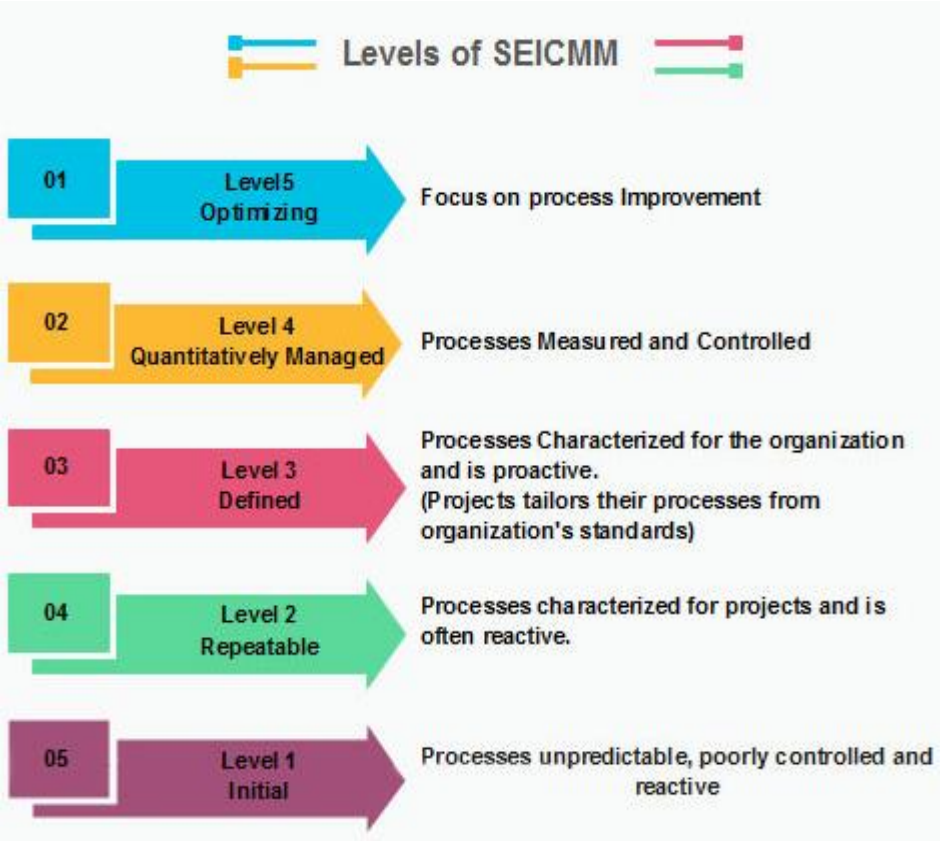
There are two methods of SEI-CMM:



Capability Evaluation: Capability evaluation provides a way to assess the software process capability of an organization. The results of capability evaluation indicate the likely contractor performance if the contractor is awarded a work. Therefore, the results of the software process capability assessment can be used to select a contractor.

Software Process Assessment: Software process assessment is used by an organization to improve its process capability. Thus, this type of evaluation is for purely internal use.

SEI CMM categorized software development industries into the following five maturity levels. The various levels of SEI CMM have been designed so that it is easy for an organization to build its quality system starting from scratch slowly.



Level 1: Initial

Ad hoc activities characterize a software development organization at this level. Very few or no processes are described and followed. Since software production processes are not limited, different engineers follow their process and as a result, development efforts become chaotic. Therefore, it is also called a chaotic level.

Level 2: Repeatable

At this level, the fundamental project management practices like tracking cost and schedule are established. Size and cost estimation methods, like function point analysis, COCOMO, etc. are used.

Level 3: Defined

At this level, the methods for both management and development activities are defined and documented. There is a common organization-wide understanding of operations, roles, and responsibilities. The ways through defined, the process and product qualities are not measured. ISO 9000 goals at achieving this level.

Level 4: Managed

At this level, the focus is on software metrics. Two kinds of metrics are composed.

Product metrics measure the features of the product being developed, such as its size, reliability, time complexity, understandability, etc.

Process metrics follow the effectiveness of the process being used, such as average defect correction time, productivity, the average number of defects found per hour inspection, the average number of failures detected during testing per LOC, etc. The software process and product quality are measured, and quantitative quality requirements for the product are met. Various tools like Pareto charts, fishbone diagrams, etc. are used to measure the product and process quality. The process metrics are used to analyze if a project performed satisfactorily. Thus, the outcome of process measurements is used to calculate project performance rather than improve the process.

Level 5: Optimizing

At this phase, process and product metrics are collected. Process and product measurement data are evaluated for continuous process improvement.

Key Process Areas (KPA) of a software organization

Except for SEI CMM level 1, each maturity level is featured by several Key Process Areas (KPAs) that contains the areas an organization should focus on improving its software process to the next level. The focus of each level and the corresponding key process areas are shown in the fig.

CMM Level	Focus	Key Process Areas
1. Initial	Competent People	NO KPA'S
2. Repeatable	Project Management	Software Project Planning software Configuration Management
3. Defined	Definition of Processes	Process definition Training Program Peer reviews
4. Managed	Product and Process quality	Quantitiative Process Metrics Software Quality Management
5. Optimizing	Continuous Process improvement	Defect Prevention Process change management Technology change management

SEI CMM provides a series of key areas on which to focus to take an organization from one level of maturity to the next. Thus, it provides a method for gradual quality improvement over various stages. Each step has been carefully designed such that one step enhances the capability already built up.

Q.11 Discuss the importance of Feasibility Study. Also discuss its various types**. (AKTU 2023-24)**

Ans. A **Feasibility Study** is a critical analysis used to assess the practicality and viability of a proposed project or idea. It evaluates whether the project can be successfully executed, identifying potential risks, costs, and benefits. The goal is to determine whether the project is worthwhile and if it is likely to meet the objectives within available resources and constraints. Feasibility studies play a crucial role in decision-making by providing stakeholders with clear insights and information to minimize uncertainties before committing to a project.

Importance of Feasibility Study

1. **Risk Reduction**: It helps identify potential risks, challenges, and obstacles that may arise during the project. By recognizing these factors early, the project can be adjusted or abandoned before resources are overly invested.
2. **Cost and Resource Assessment**: It provides a detailed cost estimate, allowing stakeholders to determine if the financial, human, and material resources required for the project are available and sustainable.
3. **Decision-Making Tool**: It helps businesses or project owners decide whether to proceed with a project. Without a feasibility study, projects may fail due to unforeseen difficulties, lack of resources, or poor planning.
4. **Market and Demand Analysis**: A feasibility study evaluates market demand and competition, which helps in understanding whether there is sufficient demand for the product or service.

5. **Timeframe and Scheduling**: It assesses whether the project can be completed within the desired timeframe. A realistic timeline is essential for maintaining momentum and meeting deadlines.

Types of Feasibility Studies

1. **Technical Feasibility**:
   This type assesses whether the organization has the technical expertise, equipment, and resources required to complete the project. It explores questions like:
   - Do we have the technology and infrastructure to implement the project?
   - Is the technology up to date?
   - Can the required systems and software work together?
2. **Economic Feasibility**:
   This type focuses on the financial aspects of the project. It evaluates whether the project is financially viable and whether the projected benefits outweigh the costs. Key components include:
   - Project costs (initial, operating, and maintenance)
   - Return on Investment (ROI)
   - Break-even analysis
   - Financial risks
3. **Legal Feasibility**:
   This type examines whether the project complies with legal, regulatory, and environmental requirements. It assesses:
   - Local, state, and federal regulations
   - Zoning laws, permits, and licenses
   - Environmental regulations
   - Intellectual property rights and contracts
4. **Operational Feasibility**:
   This type evaluates whether the project can be successfully integrated into the existing operational environment of the organization. It involves:
   - Assessing the internal processes
   - Understanding the organization's ability to adapt to new systems or changes
   - Identifying potential workflow changes or improvements
   - Determining the operational resources required (staffing, training, etc.)
5. **Market Feasibility**:
   This type focuses on the demand for the product or service, as well as market competition and customer interest. It involves:
   - Market analysis and segmentation
   - Competitor analysis
   - Identifying target customers and understanding their needs
   - Projecting future demand and trends
6. **Schedule Feasibility**:
   This type assesses whether the project can be completed within the proposed timeline. It involves:
   - Estimating project duration
   - Identifying key milestones and deadlines
   - Assessing any constraints that might delay the project

7. **Cultural and Social Feasibility**:
   This type assesses how well the project aligns with the cultural, social, and organizational environment. It explores:
   - Public acceptance of the project
   - Social impact and consequences
   - Stakeholder interests and concerns

Q.12 Explain Requirement Elicitation techniques in detail. **(AKTU 2023-24)**

Ans. Requirement Elicitation is the process of collecting the requirements or expectations of a project, system, or product from stakeholders, users, and other sources. It's a crucial step in the software development life cycle, as well as in any other project management or product development process. Effective elicitation helps ensure that the final deliverable aligns with user needs, business goals, and system constraints.

Several techniques are used during the requirements elicitation phase to gather detailed and accurate information. Each technique has its strengths and is applicable in different contexts depending on the nature of the project, the stakeholders involved, and the available resources.

Here's a detailed explanation of the most common **requirement elicitation techniques**:

1. Interviews

Interviews involve direct one-on-one or group conversations between the requirements analyst and stakeholders (such as users, clients, or subject matter experts). This technique allows the analyst to gather insights, clarify expectations, and explore requirements in depth.

- **Structured Interviews**: Follow a predefined set of questions. They are useful for gathering specific, detailed information.
- **Unstructured Interviews**: More conversational and flexible, allowing the participant to express ideas freely. These can reveal insights and uncover unknown requirements.
- **Semi-Structured Interviews**: A mix of both structured and unstructured, where the interviewer has a general guide but can explore topics further as they arise.

2. Workshops

Workshops are facilitated group sessions where stakeholders and team members come together to discuss requirements. These sessions may involve brainstorming, prioritization, and collaborative exercises to define the requirements.

- **JAD (Joint Application Development)**: A structured workshop where developers, users, and analysts collaborate to define system requirements, usually in a short time.
- **Brainstorming**: Group participants generate ideas freely, which can then be sorted and refined.
- **Focus Groups**: A small group of participants discuss their needs and expectations to provide a clear set of requirements.

3. Surveys/Questionnaires

Surveys and Questionnaires are written forms that are distributed to stakeholders or end-users to collect their input on system requirements. This technique is particularly useful when there are many participants, or when you need to collect data from a large user base quickly.

- **Closed-Ended Questions**: Respondents choose from predefined options (yes/no, multiple choice).
- **Open-Ended Questions**: Respondents provide their own answers in their own words.

4. Observations

Observations involve watching users interact with a system or process in their natural environment. This method allows the analyst to gather data on actual behavior, often revealing requirements that users might not articulate in interviews or surveys.

- **Direct Observation**: The analyst observes the users while they perform tasks.
- **Shadowing**: The analyst follows and records the user's actions closely over a period of time.

5. Document Analysis

Document Analysis involves reviewing existing documentation such as business process models, technical specifications, user manuals, and previous project documents to extract relevant information for requirements.

- **Historical Documents**: Reviewing past projects or systems to identify similar requirements.
- **Existing System Analysis**: Understanding the current system's capabilities and limitations, which can inform the requirements for a new system.

6. Use Cases

Use Case Analysis involves defining detailed scenarios of how users will interact with a system. Use cases describe specific interactions between users (actors) and the system to achieve a particular goal.

- **Functional Use Cases**: Describes the expected functionality.
- **System Use Cases**: Outlines how the system will support the user’s interaction.

7. Prototyping

Prototyping involves creating a working model (prototype) of the system or part of the system and allowing users to interact with it. This iterative process helps users clarify and refine their requirements by testing and providing feedback on early versions.

- **Throwaway Prototyping**: Develop a prototype quickly to gather feedback, then discard it.
- **Evolutionary Prototyping**: Develop a prototype and refine it over time, improving the system based on feedback.

8. Brainstorming

Brainstorming involves a group of people generating ideas or requirements spontaneously in an unstructured setting. It encourages creativity and the free flow of ideas without immediate criticism or evaluation.

- Typically done in small groups.
- Ideas are noted down for further evaluation and prioritization.

9. Storyboarding

Storyboarding is a visual elicitation technique where scenarios or system interactions are presented as a series of visual panels or slides. These visual representations help stakeholders visualize and articulate requirements in a more interactive and engaging way.

- Similar to the storyboarding used in movies, but focused on system interactions.

BUDDHA SERIES

(Unit Wise Solved Question & Answers)

Course – B.Tech. (CSE)

College – Buddha Institute of Technology

(AKTU CODE-525)

Department: Computer Science & Engineering(AIML/DS)

Subject: Software Engineering (BCS 601)

Faculty Name: Mr. Satish Kumar

Unit - 3

Q.1 Differentiate between cohesion & coupling. **(AKTU 2018-19)**
Ans:

Cohesion	Coupling
Cohesion is the indication of the relationship within the module.	Coupling is the indication of the relationships between modules.
Cohesion shows the module's relative functional strength.	Coupling shows the relative independence among the modules.
Cohesion is a degree (quality) to which a / module focuses on a single thing.	Coupling is the degree to which a component/module is connected to the other modules.
While designing you should strive for high cohesion i.e. a cohesive component/ module focus on a single task (i.e., single-mindedness) with little interaction with other modules of the system.	While designing you should strive for low coupling i.e. dependency between modules should be less.
Cohesion is the kind of natural extension of data hiding, for example, a class having all members visible with a package having default visibility.	Making private fields, private methods and non-public classes provide loose coupling.
Cohesion is Intra -Module Concept.	Coupling is Inter -Module Concept.

Q.2 Write the difference between function oriented design & object oriented design.
(AKTU 2018-19)

Ans: Function-Oriented Design (FOD) and Object-Oriented Design (OOD) are two different paradigms used in software engineering to structure and organize software systems. Each approach has its own principles, methodologies, and benefits. Here are the key differences between the two:

Function-Oriented Design (FOD)

Basic Concept:

Focuses on the functions or procedures that the software needs to perform.

The system is designed around a set of high-level functions that process data.

Decomposition:

Decomposes the system into smaller, more manageable functions or procedures.

Functions are grouped based on the tasks they perform.

Data Handling:

Data and functions are separate entities.

Global data structures are often used and shared among functions.

Modularity:

Emphasizes the modularity of functions.

Each function performs a specific task and can be reused in different parts of the system.

Approach:

Top-down approach is commonly used, starting with a high-level overview and breaking it down into smaller functions.

Design Representation:

Uses data flow diagrams (DFDs) and structure charts to represent the system design.

Reusability:

Functions can be reused, but reusability is often limited due to the dependence on global data **structures.**

Examples:

Procedural programming languages such as C, Fortran, and Pascal.

Object-Oriented Design (OOD)

Basic Concept:

Focuses on objects that represent both data and behavior.

The system is designed around objects that interact with one another.

Decomposition:

Decomposes the system into objects that encapsulate both data and behavior.

Objects are grouped based on their roles and responsibilities.

Data Handling:

Data and functions (methods) are encapsulated within objects.

Each object manages its own state and behavior.

Modularity:

Emphasizes the modularity of objects.

Objects can be easily reused and extended due to encapsulation and inheritance.

Approach:

Both top-down and bottom-up approaches can be used.

Design is often iterative and incremental.

Design Representation:

Uses Unified Modeling Language (UML) diagrams such as class diagrams, sequence diagrams, and use case diagrams to represent the system design.

Reusability:

High reusability due to encapsulation, inheritance, and polymorphism.

Objects can be easily reused in different systems or contexts.

Examples:

Object-oriented programming languages such as Java, C++, Python, and C#.

COMPARISON FACTORS	FUNCTION ORIENTED DESIGN	OBJECT ORIENTED DESIGN
Abstraction	The basic abstractions, which are given to the user, are real world functions.	The basic abstractions are not the real world functions but are the data abstraction where the real world entities are represented.
Function	Functions are grouped together by which a higher level function is obtained.	Functions are grouped together on the basis of the data they operate since the classes are associated with their methods.

Execute	carried out using structured analysis and structured design i.e, data flow diagram	Carried out using UML
State information	In this approach the state information is often represented in a centralized shared memory.	In this approach the state information is not represented is not represented in a centralized memory but is implemented or distributed among the objects of the system.
Approach	It is a top down approach.	It is a bottom up approach.
Begins basis	Begins by considering the use case diagrams and the scenarios.	Begins by identifying objects and classes.
Decompose	In function oriented design we decompose in function/procedure level.	We decompose in class level.
Use	This approach is mainly used for computation sensitive application.	This approach is mainly used for evolving system which mimics a business or business case.

Q.3 Compute the function point value for a project with the following information domain characteristics: Number of user inputs = 30
Number of user outputs = 42
Number of user enquiries = 8
Number of files = 7
Number of extended interfaces = 6 **(AKTU 2021-22)**
Ans:

Step 1:-

As complexity adjustment factor/value is moderate (given in question), Hence.

Scale = 2

F = 14×2 = 28

Step-2:-

CAF = 0.65 + (0.01) (Σfi)

CAF = 0.65 + (0.01×28)

CAF = 0.93

EI = 30×3 + 30×4 + 30×6 = 390

EO = 42×4 + 42×5 + 48×7 = 672

EQ = 8×3 + 8×4 + 8×6 = 104

ILF = 7×4 + 7×10 + 7×15 = 224

EIF = 6×5 + 6×7 + 6×10 = 132

UFP = 390 + 672 + 104 + 224 + 132 = 1522

Functional point = UFP * CAF

= 1522 × 0.93

= 1415.46 Ans

Q.4 Illustrate the principles of software design. Discuss the characteristics of good software design. **(AKTU 2022-23)**

Ans: Software design is a critical phase in software development, bridging the gap between requirements analysis and implementation. It involves defining the architecture, components, interfaces, and data for a system to satisfy specified requirements. Good software design is essential for creating reliable, maintainable, and scalable software. Here are the key principles and characteristics of good software design:

Principles of Software Design

Modularity

Definition: Breaking down a software system into smaller, self-contained units (modules) that can be developed, tested, and understood independently.

Benefit: Enhances maintainability and reusability, and simplifies debugging and testing.

Abstraction

Definition: Simplifying complex reality by modeling classes appropriate to the problem and working at the most relevant level of detail.

Benefit: Hides the complex implementation details, exposing only the necessary parts, making the system easier to understand and manage.

Encapsulation

Definition: Bundling the data (attributes) and the methods (functions) that operate on the data into a single unit, usually a class, and restricting access to some of the object's components.

Benefit: Protects the integrity of the data by preventing outside interference and misuse.

Separation of Concerns

Definition: Dividing a software system into distinct features that overlap in functionality as little as possible.

Benefit: Reduces complexity, improves code readability, and makes the system easier to manage and modify.

Single Responsibility Principle (SRP)

Definition: Each module or class should have only one reason to change, meaning it should only have one job or responsibility.

Benefit: Simplifies testing and debugging, as changes in one part of the system are less likely to affect other parts.

Open/Closed Principle

Definition: Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

Benefit: Enhances flexibility and minimizes the risk of introducing errors when new functionalities are added.

Liskov Substitution Principle

Definition: Objects of a superclass should be replaceable with objects of a subclass without affecting the functionality of the program.

Benefit: Ensures that a subclass can stand in for its superclass without causing errors or unexpected behavior.

Interface Segregation Principle

Definition: No client should be forced to depend on methods it does not use. Multiple specific interfaces are better than one general-purpose interface.

Benefit: Improves code readability and reduces the impact of changes.

Dependency Inversion Principle

Definition: High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

Benefit: Enhances flexibility and decouples the system's components, making it easier to manage

and extend.

Characteristics of Good Software Design

Correctness

Definition: The software meets all specified requirements and performs its intended functions accurately.

Benefit: Ensures the system delivers the expected outcomes.

Maintainability

Definition: The ease with which software can be modified to correct faults, improve performance, or adapt to a changed environment.

Benefit: Reduces the time and effort required to make changes, ensuring long-term sustainability.

Reusability

Definition: The ability of software components to be used in more than one system or in building other applications.

Benefit: Saves development time and costs by leveraging existing components.

Scalability

Definition: The ability of the software to handle increased loads or to be readily enlarged.

Benefit: Ensures the system can grow and adapt to increased demands without a complete redesign.

Performance

Definition: The system performs its tasks within acceptable time frames and resource usage.

Benefit: Enhances user satisfaction and system efficiency.

Security

Definition: The system protects against unauthorized access and ensures data integrity and confidentiality.

Benefit: Safeguards sensitive information and maintains user trust.

Robustness

Definition: The software can handle unexpected inputs or situations without crashing or producing incorrect results.

Benefit: Enhances the system's reliability and user confidence.

Usability

Definition: The system is user-friendly and easy to navigate, allowing users to achieve their goals efficiently.

Benefit: Improves user experience and satisfaction.

Portability

Definition: The software can be easily transferred from one environment to another.

Benefit: Enhances the flexibility of the software to operate on different platforms and environments.

Testability

Definition: The software facilitates easy testing to ensure all parts work as intended.

Benefit: Reduces the likelihood of defects and increases the reliability of the software.

In summary, good software design follows established principles to ensure that the final product is reliable, efficient, and adaptable. By adhering to these principles and aiming for these characteristics, software engineers can create systems that are not only functional but also sustainable and user-friendly.

Q.5 Assume that all complexity adjustment values are moderate. **(AKTU 2021-22)**

```
if (t<b)
{
        t = t+3;
        t++;
}
else if (t > 10)
{
        t--;
}
else
{
        t++;
  }
```

Compute N,V,D,E & T using Halstead's Software Science.

Ans.

FU	Weighting factors		
	Low	Avg	High
EI	3	4	6
EO	4	5	7
EQ	3	4	6
ILF	7	10	15
EIF	5	7	10

Solve :-

Operators	No. of occurance	Operands	No. of occurance
if	2	t	7
( )	2	6	1
<	1	3	1
{ }	3	10	1
=	1		
+	1		
++	2		
;	4		
else	2		
>	1		
--	1		
$n_1 = 11$	$N_1 = 20$	$n_2 = 4$	$N_2 = 10$

(i) Total length (N) = $N_1 + N_2$ = $N = 30$

(ii) Vocabulary (n) = $n_1 + n_2 = 11 + 4 = 15$

(iii) Volume (V) = $N \log_2 n = 30 \log_2 15 = 35.2$

(iv) $D = n_1 \times N_2 / 2 \times n_2 = 11 \times 10 / 2 \times 4 = 110/8 = 13.75$

(v) $E = D \times V = 13.75 \times 35.2 = 484$

(vi) $B = V/3000 = 35.2/3000 = 0.011$

(vii) $T = \frac{E}{18} = \frac{484}{18} = 26.88$ Sec

Q.6 Explain Software Design. Write down the various procedures for design phase.

(AKTU 2018-19)

Ans: Software design is a critical phase in the software development lifecycle (SDLC) that involves planning and defining the architecture, components, interfaces, and data for a system to ensure it meets specified requirements. This phase translates requirements into a blueprint for constructing the software, guiding developers and serving as a reference throughout the development process.

Procedures for the Design Phase

The design phase typically involves several key procedures, each contributing to the overall design of the software system. Here's an outline of the primary procedures involved in the software design phase.

Requirement Analysis Review:

Purpose: Ensure that all requirements are fully understood and any ambiguities are resolved.

Activities:

Review requirement documents.

Clarify any uncertainties with stakeholders.

Identify key functional and non-functional requirements.

High-Level Design (HLD):

Purpose: Define the overall architecture and structure of the software system.

Activities:

Architectural Design: Define the system's architecture, including the main components and their interactions.

Module Decomposition: Break down the system into smaller modules or subsystems.

Technology Selection: Decide on the technology stack, including programming languages, frameworks, and tools.

Data Design: Outline the data structures and database schemas.

Interface Design: Define interfaces between modules and with external systems.

Low-Level Design (LLD):

Purpose: Provide detailed design specifications for each component or module.

Activities:

Class Diagrams: Create detailed class diagrams if using an object-oriented approach.

Algorithm Design: Define the algorithms to be used in the modules.

Pseudo code/Flowcharts: Write pseudocode or draw flowcharts to describe the logic of individual modules.

Data Flow Diagrams (DFD): Develop detailed DFDs to illustrate data movement within the system.

State Diagrams: Use state diagrams to model the dynamic behavior of the system.

User Interface (UI) Design:

Purpose: Create the layout and design of the user interface.

Activities:

Wireframes: Develop wireframes to outline the basic structure and layout of the UI.

Prototypes: Create interactive prototypes to visualize and test the UI design.

Usability Testing: Conduct usability testing to ensure the interface is user-friendly and meets user needs.

Design Guidelines: Define UI design guidelines and standards to maintain consistency.

Component Design:

Purpose: Design each individual component or module in detail.

Activities:

Component Specification: Define the responsibilities, interfaces, and dependencies of each component.

Detailed Diagrams: Create detailed diagrams (e.g., UML diagrams) to illustrate the internal structure of components.

Error Handling: Design error handling and recovery mechanisms.

Database Design:

Purpose: Design the structure of the database that will store the system's data.

Activities:

ER Diagrams: Create Entity-Relationship diagrams to model data entities and their relationships.

Normalization: Normalize the database to eliminate redundancy and ensure data integrity.

Schema Design: Define database schemas, including tables, fields, indexes, and relationships.

Security Design:

Purpose: Ensure the system is secure and protected against threats.

Activities:

Security Requirements: Identify and document security requirements.

Threat Modeling: Analyze potential threats and define countermeasures.

Access Control: Design access control mechanisms to restrict unauthorized access.

Data Encryption: Specify methods for encrypting sensitive data.

Design Documentation:

Purpose: Create comprehensive documentation of the design for reference and implementation.

Activities:

Design Specifications: Write detailed design specifications for all aspects of the system.

Diagrams and Models: Include all relevant diagrams, models, and schemas.

Rationale: Document the rationale behind design decisions.

Review and Approval: Conduct design reviews and obtain approval from stakeholders.

Conclusion

The design phase is crucial for laying a solid foundation for software development. It involves a series of structured procedures that ensure all aspects of the system are well-thought-out and documented. Effective software design not only guides developers during implementation but also helps in maintaining the system, ensuring it is scalable, reliable, and meets user expectations.

Q.7 Explain Halstead software metrics in detail and mention what do you understand by token count? **(AKTU 2018-19)**

Ans: Halstead Software Metrics

Halstead Software Metrics, developed by Maurice Halstead in 1977, are a set of quantitative measures that aim to estimate various attributes of software, such as complexity, effort, and maintainability. These metrics are derived from the source code's lexical properties, focusing on the number of operators and operands. The fundamental premise is that software characteristics can be predicted using these counts.

Key Concepts of Halstead Metrics

Operators and Operands:

Operators (n1): Symbols or keywords that denote operations, such as +, -, *, /, if, while, etc.

Operands (n2): Identifiers or values that operators manipulate, such as variables, constants, and function names.

Vocabulary and Length:

Vocabulary (n): The total number of unique operators and operands.

$$n = n1 + n2$$

Length (N): The total number of occurrences of operators and operands.

$$N = N1 + N2$$

where N1 is the total occurrences of operators, and N2 is the total occurrences of operands.

Volume (V): Measures the size of the implementation of the algorithm.

$$V = N \times \log_2(n)$$

Difficulty (D): Indicates how challenging the program is to write or understand.

$$D = \frac{n1}{2} \times \frac{N2}{n2}$$

Effort (E): Represents the mental effort required to develop or maintain the software.

$$E = D \times V$$

Time (T): The time required to program, calculated based on effort.

$$T = \frac{E}{18}$$

Bugs (B): An estimate of the number of potential errors in the implementation.

$$B = \frac{E^{2/3}}{3000}$$

Detailed Explanation of Halstead Metrics

Vocabulary (n): The unique count of operators and operands gives an insight into the program's lexical diversity. A higher vocabulary suggests more varied use of language constructs.

Length (N): The total count of operators and operands shows how many language elements are used in total. This helps in understanding the size and complexity of the code.

Volume (V): Volume provides a measure of the code's size in terms of the number of mental discriminations needed to understand the program. It's essentially a measure of the information

content of the program.

Difficulty (D): Difficulty indicates the complexity of understanding and maintaining the code. It factors in how operators and operands are distributed, suggesting the intellectual effort needed.

Effort (E): Effort combines volume and difficulty to estimate the total amount of work required to write or understand the program. This can be used to estimate the time and cost of the software project.

Time (T): Time is derived from effort and provides a rough estimate of the actual time required to implement the code, assuming a standard rate of mental processing.

Bugs (B): Bugs give an estimation of the number of errors that might be present in the code, helping in risk assessment and planning for debugging and testing.

Token Count

Token count refers to the total number of distinct elements in the source code, which are categorized into operators and operands. In the context of Halstead metrics:

Operators: Symbols that perform actions or control structures (e.g., arithmetic operators, logical operators, keywords like if, else, for, etc.).

Operands: Variables, constants, and identifiers upon which operations are performed.

Operators (n1): {int, add, (,), return, +, ;, {, }}

Unique operators (n1) = 9

Total operators (N1) = 10 (counting each occurrence)

Operands (n2): {a, b}

Unique operands (n2) = 2

Total operands (N2) = 4 (counting each occurrence)

Vocabulary (n):

$$n = n1 + n2 = 9 + 2 = 11$$

Length (N):

$$N = N1 + N2 = 10 + 4 = 14$$

Volume (V):

$$V = N \times \log_2(n) = 14 \times \log_2(11) \approx 14 \times 3.46 = 48.44$$

Difficulty (D):

$$D = \frac{n1}{2} \times \frac{N2}{n2} = \frac{9}{2} \times \frac{4}{2} = 4.5 \times 2 = 9$$

Effort (E):

$$E = D \times V = 9 \times 48.44 = 435.96$$

Time (T):

$$T = \frac{E}{18} = \frac{435.96}{18} \approx 24.22 \text{ seconds}$$

Bugs (B):

$$B = \frac{E^{2/3}}{3000} \approx \frac{(435.96)^{2/3}}{3000} \approx \frac{49.15}{3000} \approx 0.016$$

This example shows how Halstead metrics provide quantitative insights into the software's complexity, effort, and potential error rate, aiding in better understanding and managing the software development process.

Q.8 Differentiate between the features of Top-down and Bottom-up approaches of software design along with its advantages and disadvantages. **(AKTU 2021-22)**

Ans:

TOP DOWN APPROACH	BOTTOM UP APPROACH
In this approach We focus on breaking up the problem into smaller parts.	In bottom up approach, we solve smaller problems and integrate it as whole and complete the solution.
Mainly used by structured programming language such as COBOL, Fortran, C, etc.	Mainly used by object oriented programming language such as C++, C#, Python.
Each part is programmed separately therefore contain redundancy.	Redundancy is minimized by using data encapsulation and data hiding.

TOP DOWN APPROACH	BOTTOM UP APPROACH
In this the communications is less among modules.	In this module must have communication.
It is used in debugging, module documentation, etc.	It is basically used in testing.
In top down approach, decomposition takes place.	In bottom up approach composition takes place.
In this top function of system might be hard to identify.	In this sometimes we cannot build a program from the piece we have started.
In this implementation details may differ.	This is not natural for people to assemble.
Pros- • Easier isolation of interface errors • It benefits in the case error occurs towards the top of the program. • Defects in design get detected early and can be corrected as an early working module of the program is available.	**Pros-** • Easy to create test conditions • Test results are easy to observe • It is suited if defects occur at the bottom of the program.
Cons- • Difficulty in observing the output of test case. • Stub writing is quite crucial as it leads to setting of output parameters. • When stubs are located far from the top level module, choosing test cases and designing stubs become more challenging.	**Cons-** • There is no representation of the working model once several modules have been constructed. • There is no existence of the program as an entity without the addition of the last module. • From a partially integrated system, test engineers cannot observe system-level functions. It can be possible only with the installation of the top-level test driver.

Q.9 What is the software architecture? **(AKTU 2022-23)**

Ans: Software Architecture

Definition: Software architecture refers to the high-level structure of a software system, encompassing the organization of its components, the relationships between those components, and the principles and guidelines governing their design and evolution. It serves as a blueprint for both the system and the project, laying out the framework for constructing the software.

Key Elements of Software Architecture

Components:

These are the individual units of functionality within the system. Components can be classes, modules, services, or any other discrete unit of software that performs a specific function.

Connectors:

Connectors define the interactions between components. They manage communication and coordination, including method calls, data exchange, and synchronization.

Configurations:

Configurations represent the overall structure of the system, showing how components and connectors are arranged and interact with each other.

Architectural Styles/Patterns:

These are standard solutions to common architectural problems. Examples include Layered Architecture, Microservices, Client-Server, Event-Driven, and Service-Oriented Architecture (SOA).

Quality Attributes:

These are the non-functional requirements that the architecture must support, such as performance, scalability, security, maintainability, and usability.

Importance of Software Architecture

Foundation for Development:

Provides a structured solution that meets all technical and operational requirements.

Serves as a blueprint for both the system and the project, guiding development and evolution.

Manage Complexity:

Simplifies the management of system complexity by breaking down the system into manageable components and defining clear interfaces and interactions.

Facilitate Communication:

Serves as a common language among stakeholders, including developers, managers, and customers, ensuring everyone has a shared understanding of the system.

Ensure Quality:

Addresses key quality attributes like performance, security, and maintainability from the outset, rather than as an afterthought.

Support Evolution:

Provides a flexible framework that can accommodate future changes and extensions, ensuring the system can evolve over time.

Architectural Patterns

Layered Architecture:

Divides the system into layers, each with specific responsibilities. Common layers include presentation, business logic, and data access.

Advantages: Separation of concerns, ease of maintenance, and scalability.

Disadvantages: Can introduce latency and complexity in coordination between layers.

Microservices:

Structures the system as a collection of loosely coupled, independently deployable services.

Advantages: Scalability, resilience, and ease of deployment.

Disadvantages: Increased complexity in management and communication between services.

Client-Server:

Divides the system into two parts: clients that request services and servers that provide services.

Advantages: Centralized control, ease of management, and scalability.

Disadvantages: Server bottlenecks and single points of failure.

Event-Driven Architecture:

Components communicate through events, which are messages indicating that something has happened.

Advantages: High decoupling, scalability, and responsiveness.

Disadvantages: Complexity in managing event flows and ensuring consistency.

Service-Oriented Architecture (SOA):

Organizes the system as a collection of services that communicate over a network.

Advantages: Reusability, flexibility, and integration with other systems.

Disadvantages: Complexity in managing services and ensuring reliable communication.

Steps in Designing Software Architecture

Requirement Analysis:

Gather and analyze both functional and non-functional requirements to understand the needs of the system.

Architectural Design:

Define the high-level structure, including components, connectors, and configurations.

Choose appropriate architectural styles and patterns.

Modeling:

Create architectural models using diagrams (e.g., UML diagrams) to represent components, relationships, and interactions.

Evaluation:

Assess the architecture against requirements and quality attributes using techniques like ATAM (Architecture Tradeoff Analysis Method).

Documentation:

Document the architecture, including diagrams, rationale for decisions, and guidelines for implementation.

Implementation and Maintenance:

Guide the development team in implementing the architecture.

Continuously refine and evolve the architecture to meet changing requirements and improve

quality.

Example

Consider an e-commerce application using a Microservices architecture:

Components:

User Service, Product Service, Order Service, Payment Service, Notification Service.

Connectors:

REST APIs for synchronous communication.

Message queues for asynchronous communication.

Configuration:

Each service runs in its container, orchestrated using Kubernetes.

Quality Attributes:

Scalability: Services can be scaled independently.

Resilience: Failures in one service do not affect others.

Maintainability: Services can be updated and deployed independently.

Conclusion

Software architecture is a fundamental aspect of software engineering that defines the structure and behavior of a software system. By providing a high-level blueprint, it guides development, ensures quality, and facilitates communication and understanding among stakeholders. Effective architecture design is crucial for building scalable, maintainable, and robust software systems.

Q.10 Explain Halstead software metrics in detail and mention what do you understand by token count? **(AKTU 2021-22)**

Ans: Halstead Software Metrics

Halstead Software Metrics, developed by Maurice Halstead in 1977, are a set of quantitative measures that aim to estimate various attributes of software, such as complexity, effort, and maintainability. These metrics are derived from the source code's lexical properties, focusing on the number of operators and operands. The fundamental premise is that software characteristics can be predicted using these counts.

Key Concepts of Halstead Metrics

Operators and Operands:

Operators (n1): Symbols or keywords that denote operations, such as +, -, *, /, if, while, etc.

Operands (n2): Identifiers or values that operators manipulate, such as variables, constants, and function names.

Vocabulary and Length:

Vocabulary (n): The total number of unique operators and operands.

$$n = n1 + n2$$

Length (N): The total number of occurrences of operators and operands.

$$N = N1 + N2$$

where N1 is the total occurrences of operators, and N2 is the total occurrences of operands.

Volume (V): Measures the size of the implementation of the algorithm.

$$V = N \times \log_2(n)$$

Difficulty (D): Indicates how challenging the program is to write or understand.

$$D = \frac{n1}{2} \times \frac{N2}{n2}$$

Effort (E): Represents the mental effort required to develop or maintain the software.

$$E = D \times V$$

Time (T): The time required to program, calculated based on effort.

$$T = \frac{E}{18}$$

Bugs (B): An estimate of the number of potential errors in the implementation.

$$B = \frac{E^{2/3}}{3000}$$

Detailed Explanation of Halstead Metrics

Vocabulary (n): The unique count of operators and operands gives an insight into the program's lexical diversity. A higher vocabulary suggests more varied use of language constructs.

Length (N): The total count of operators and operands shows how many language elements are used in total. This helps in understanding the size and complexity of the code.

Volume (V): Volume provides a measure of the code's size in terms of the number of mental discriminations needed to understand the program. It's essentially a measure of the information content of the program.

Difficulty (D): Difficulty indicates the complexity of understanding and maintaining the code. It factors in how operators and operands are distributed, suggesting the intellectual effort needed.

Effort (E): Effort combines volume and difficulty to estimate the total amount of work required

to write or understand the program. This can be used to estimate the time and cost of the software project.

Time (T): Time is derived from effort and provides a rough estimate of the actual time required to implement the code, assuming a standard rate of mental processing.

Bugs (B): Bugs give an estimation of the number of errors that might be present in the code, helping in risk assessment and planning for debugging and testing.

Token Count

Token count refers to the total number of distinct elements in the source code, which are categorized into operators and operands. In the context of Halstead metrics:

Operators: Symbols that perform actions or control structures (e.g., arithmetic operators, logical operators, keywords like if, else, for, etc.).

Operands: Variables, constants, and identifiers upon which operations are performed.

Q.11 Explain software metric? Also explain the various metrics for the size estimation of a project**. (AKTU 2023-24)**

Ans. A software metric is a standard of measurement used to assess the attributes or performance of software development processes, products, or activities. These metrics provide quantitative data that help in understanding, controlling, and improving various aspects of software development, such as quality, productivity, and efficiency.

Metrics in software development are important for:

1. **Performance Monitoring:** Tracking the progress of a project or individual tasks**.**
2. **Quality Assurance:** Evaluating the effectiveness and reliability of the software.
3. **Resource Management:** Estimating costs, time, and efforts required for project completion.
4. **Decision Making:** Supporting project managers and developers in making informed decisions.

Various Metrics for Size Estimation of a Project:

Size estimation is crucial in software development to predict the effort, cost, and time required to complete a project. Several metrics are used to estimate the size of a software project, and the most common ones include:

1. Lines of Code (LOC)

- **Description:** LOC is the most commonly used metric for measuring software size, and it

refers to the total number of lines in the source code, excluding comments and blank lines.

- **Usage:** This metric provides a rough estimate of the size of the software. However, it can be misleading because the complexity of the code might not always correlate with the number of lines.
- **Challenges:** It doesn't account for code efficiency or design complexity, and large projects with well-organized code can have fewer LOC but may still be complex**.**

2. Function Points (FP)

- **Description**: Function Points measure the functionality delivered by the software, focusing on the features provided to the user, such as inputs, outputs, user interfaces, and data storage.
- **Usage:** It is independent of the programming language and provides a more accurate representation of software size, especially for systems where LOC may not be an appropriate measure.
- **Challenges:** Determining the appropriate function point count requires knowledge of the software's functionality and can involve subjective judgment.

3. Use Case Points (UCP)

- **Description**: UCP is a metric used to estimate the size of the software based on the use cases defined in the system. The complexity of the use cases and actors (users or systems interacting with the software) are considered to determine the size.
- **Usage**: It's useful for object-oriented systems and projects that heavily depend on use cases.
- **Challenges**: It requires accurate identification and analysis of use cases, and it may not be ideal for systems where use cases are less defined or complex.

4. Story Points

- **Description:** Story points are an agile metric used to estimate the size of user stories or features based on their relative complexity, effort, and time required to implement them.
- **Usage**: Common in agile development methodologies like Scrum, where teams use story points to estimate the effort involved in implementing features or user stories.
- **Challenges:** Since story points are relative, they depend on the team's experience and expertise, and the results may vary between teams.

5. Kilo Bytes (KB), Mega Bytes (MB), or Gigabytes (GB)

- **Description:** These metrics estimate the size of software based on the total data size or storage requirements. This can include the size of the database, program files, documentation, etc.
- **Usage:** These measurements are more relevant when estimating the physical space the software will occupy or when considering data-heavy applications.
- **Challenges:** This metric does not measure the functional size of the software, and therefore may not correlate with the actual effort required for development.

6. Cyclomatic Complexity

- **Description:** Although not a direct size metric, cyclomatic complexity measures the number of linearly independent paths through the code, indicating the complexity of the software's control flow.
- **Usage:** It's used to gauge the complexity of the software. A higher cyclomatic complexity generally indicates more complicated code that might require more time and effort to develop and test.
- **Challenges:** It does not directly measure the size of the software but gives an indication of its complexity.

BUDDHA SERIES

(Unit Wise Solved Question & Answers)

Course – B.Tech. (CSE)

College – Buddha Institute of Technology

(AKTU CODE-525)

Department: Computer Science & Engineering(AIML/DS)

Subject: Software Engineering (BCS 601)

Faculty Name: Satish Kumar

Unit - 4

Q.1 Discuss the differences between black box and white box testing and explain how these techniques can be used simultaneously to test a system. **(AKTU 2021-22)**

Ans:

Black Box Testing	White Box Testing
It is a way of software testing in which the internal structure or the program or the code is hidden and nothing is known about it.	It is a way of testing the software in which the tester has knowledge about the internal structure or the code or the program of the software.
It is mostly done by software testers.	It is mostly done by software developers.
No knowledge of implementation is needed.	Knowledge of implementation is required.
It can be referred as outer or external software testing.	It is the inner or the internal software testing.
It is functional test of the software.	It is structural test of the software.
This testing can be initiated on the basis of requirement specifications document.	This type of testing of software is started after detail design document.
No knowledge of programming is required.	It is mandatory to have knowledge of programming.
It is the behavior testing of the software.	It is the logic testing of the software.
It is applicable to the higher levels of testing of software.	It is generally applicable to the lower levels of software testing.
It is also called closed testing.	It is also called as clear box testing.
It is least time consuming.	It is most time consuming.
It is not suitable or preferred for algorithm testing.	It is suitable for algorithm testing.
Can be done by trial and error ways and methods.	Data domains along with inner or internal boundaries can be better tested.
Example: search something on google by using keywords	**Example:** by input to check and verify loops

Using Black Box and White Box Testing Simultaneously

Combining black box and white box testing techniques can lead to more thorough and effective testing, covering both the functional and structural aspects of the software. Here's how these techniques can be used together:

Comprehensive Coverage

Black Box: Validates that the software meets user requirements and functions correctly from an external perspective.

White Box: Ensures that the internal logic and code structure are sound, optimized, and free of hidden errors.

Combined: This approach ensures that both the external functionalities and internal mechanisms are tested thoroughly, providing a higher level of confidence in the software's reliability and performance.

Sequential Testing Stages

Black Box Testing First: Begin with black box testing to validate high-level functionalities and user requirements. This can identify major defects related to the system's behavior.

White Box Testing Next: Follow with white box testing to dive into the code and verify the internal logic, finding bugs that black box testing might miss.

Iterative Process: Iterate between these stages, as fixing bugs discovered in white box testing might necessitate retesting in black box testing to ensure no new issues are introduced.

Parallel Testing

Simultaneous Execution: Conduct black box and white box tests in parallel. For instance, while developers perform unit tests (white box), QA testers can simultaneously execute functional tests (black box).

Real-Time Feedback: This parallel approach allows for real-time feedback and quicker identification of discrepancies between expected and actual behaviors.

Cross-Disciplinary Insights

Knowledge Sharing: Testers from both black box and white box perspectives can share insights. For example, a white box tester might uncover potential functional issues that a black box tester can validate against user requirements.

Holistic View: Combining insights from both techniques offers a holistic view of the software, revealing issues related to both implementation and user experience.

Practical Example of Combined Testing

Consider testing an e-commerce application:

Black Box Testing: Testers might verify that the checkout process works correctly by inputting various user details and payment information, ensuring that the process completes successfully.

White Box Testing: Developers might perform unit tests on the payment processing code to ensure all branches and conditions are handled correctly, and no security vulnerabilities exist.

Integration: After white box tests validate the underlying code, black box tests can be rerun to ensure that the overall functionality remains intact and user workflows are not disrupted by any internal code changes.

By integrating both black box and white box testing methodologies, organizations can achieve a more robust and reliable software product, addressing both functional requirements and internal code quality. This dual approach ensures that the system not only meets user expectations but also adheres to high standards of internal correctness and efficiency.

Q.2 Explain boundary value analysis and its significance with example. **(AKTU 2022-23)**

Ans: Boundary Value Analysis (BVA) is a black box testing technique that focuses on testing the boundaries or edge cases of input values rather than the central or typical values. The rationale behind BVA is that errors often occur at the boundaries of input domains rather than in the middle. By rigorously testing these boundaries, testers can uncover a significant number of defects with a relatively small number of test cases.

Significance of Boundary Value Analysis

Error Detection: Boundary values are more likely to have defects, so testing these can effectively identify errors that might not be found with other testing techniques.

Efficiency: BVA allows testers to cover a wide range of input values with fewer test cases, making the testing process more efficient and cost-effective.

Risk Reduction: By focusing on edge cases, BVA reduces the risk of defects in the most vulnerable areas of the software, leading to higher reliability and stability.

Enhanced Coverage: BVA ensures that all edge cases are tested, providing comprehensive test coverage for critical input ranges.

How Boundary Value Analysis Works

BVA involves identifying the boundaries of input ranges and then creating test cases that include:
The minimum value

Just above the minimum value

Just below the minimum value

The maximum value

Just above the maximum value

Just below the maximum value

Example of Boundary Value Analysis

Consider a software application that accepts integer inputs within the range of 1 to 100, inclusive. The goal is to test the input validation logic to ensure it correctly handles boundary values.
Steps for BVA

Identify Boundaries:

Minimum boundary: 1

Maximum boundary: 100

Create Test Cases for Boundaries:

Just below the minimum value: 0

Minimum value: 1

Just above the minimum value: 2

Just below the maximum value: 99

Maximum value: 100

Just above the maximum value: 101

Test Cases

Test Case ID	Input Value	Expected Outcome
TC01	0	Error/Invalid input
TC02	1	Valid input, proceed
TC03	2	Valid input, proceed
TC04	99	Valid input, proceed
TC05	100	Valid input, proceed
TC06	101	Error/Invalid input

Explanation

TC01: Tests the value just below the minimum boundary (0), expecting an error because it is out of the acceptable range.

TC02: Tests the minimum boundary value (1), expecting a valid input.

TC03: Tests the value just above the minimum boundary (2), expecting a valid input.

TC04: Tests the value just below the maximum boundary (99), expecting a valid input.

TC05: Tests the maximum boundary value (100), expecting a valid input.

TC06: Tests the value just above the maximum boundary (101), expecting an error because it is out of the acceptable range.

Significance in Practice

Using BVA in the example above ensures that the software correctly handles both the lower and upper edges of the input range, as well as the points just outside these edges. This testing method is particularly effective for input validation and is widely used in applications involving numerical inputs, data entry fields, and user interface controls.

By rigorously applying BVA, testers can uncover boundary-related defects early in the development cycle, enhancing the software's robustness and reliability while optimizing the testing effort.

Q.3 Write short notes on **(AKTU 2018-19)**

i) Acceptance Testing

ii) Regression Testing

Ans: Acceptance Testing

Acceptance testing is formal testing based on user requirements and function processing. It determines whether the software is conforming specified requirements and user requirements or

not. It is conducted as a kind of Black Box testing where the number of required users involved testing the acceptance level of the system. It is the fourth and last level of software testing.

User acceptance testing (UAT) is a type of testing, which is done by the customer before accepting the final product. Generally, UAT is done by the customer (domain expert) for their satisfaction, and check whether the application is working according to given business scenarios, real-time scenarios.

In this, we concentrate only on those features and scenarios which are regularly used by the customer or mostly user scenarios for the business or those scenarios which are used daily by the end-user or the customer.

However, the software has passed through three testing levels (Unit Testing, Integration Testing, System Testing) But still there are some minor errors which can be identified when the system is used by the end user in the actual scenario.

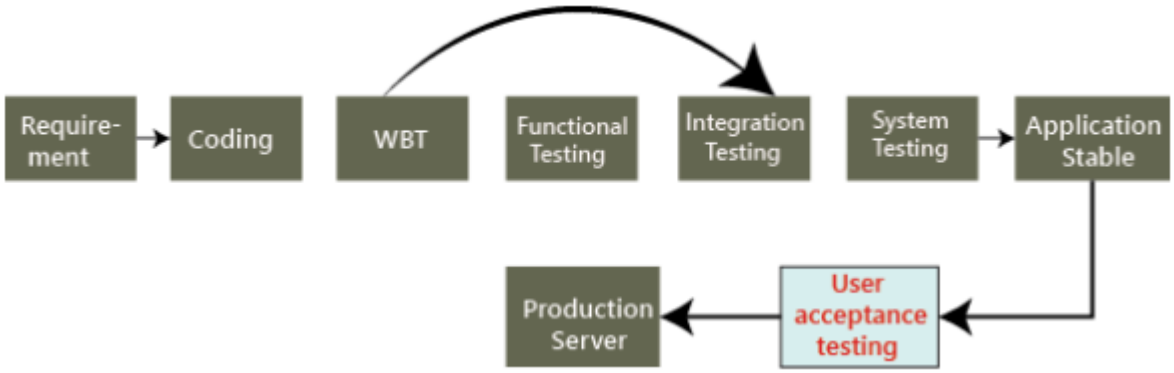
Acceptance testing is the squeezing of all the testing processes that have done previously.

Reason behind Acceptance Testing

Once the software has undergone through Unit Testing, Integration Testing and System Testing so, Acceptance Testing may seem redundant, but it is required due to the following reasons.

- During the development of a project if there are changes in requirements and it may not be communicated effectively to the development team.
- Developers develop functions by examining the requirement document on their own understanding and may not understand the actual requirements of the client.
- There's maybe some minor errors which can be identified only when the system is used by the end user in the actual scenario so, to find out these minor errors, acceptance testing is essential.

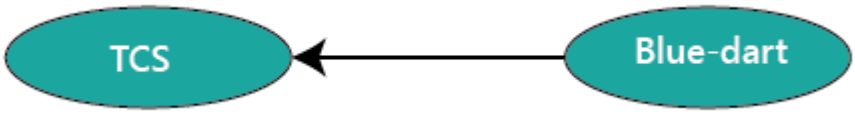
Once the application is bug-free, we handover it to the customer, no customer accept the application blindly before using it. Hence, they do one round of testing for their satisfaction, which is known as user acceptance testing.



Who performs user acceptance testing?

The acceptance testing can be performed by different persons in different cases.

For example, the blue-dart company gives the requirement to TCS for developing the application, and the TCS will accept the needs and agree to deliver the application in the two releases as we can see in the below image:

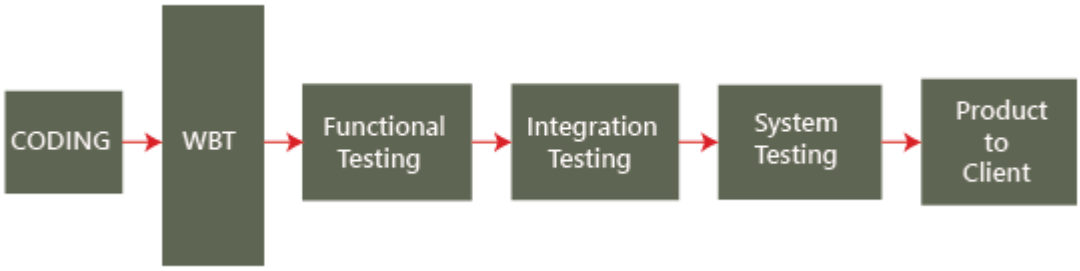


On August 10, the test manager tells the project manager that there is a critical bug in the application, and that will take another four days to fix it.

But the project manager said we have to deliver the software within a given time. It takes another 30 days to fix the defect, or otherwise, we will have to pay the penalty (fine) for each day after the given release date. Is this the real situation? NO, let us see three different cases and understand who perform the acceptance testing.

Case1

In this, we will discuss how the acceptance testing is performed, and here the test engineer will do the acceptance testing.



Mostly, the actual flow for testing the application will be seen in the above image, but here it is little difference, as we know where the end-to-end testing or system testing ends and the acceptance testing will proceed. To understand this scenario, follow the below process:

The blue-dart provides the requirements, and TCS develops the application and performs all the testing and handover to the blue-dart company.

Now the question arises the blue-dart will use the application as soon they get it from TCS? NO, the blue dart company has a group of test engineers after they get the software, and this team will start testing the application, and this end-to-end testing is done at the customer environment, which is called the **User Acceptance Testing**.

Regression Testing in Software Engineering

Definition: Regression testing is a type of software testing that ensures that recent code changes have not adversely affected existing functionalities. It verifies that previously developed and tested software still performs correctly after changes such as enhancements, patches, or configuration changes.

Purpose:

Validation: Confirm that new changes haven't introduced new bugs or reactivated old ones.

Stability: Ensure the software remains stable and reliable after updates.

Quality Assurance: Maintain the integrity and quality of the software over time.

When to Perform Regression Testing:
After bug fixes.

Following enhancements or new feature additions.

Post system upgrades or migrations.

During regular maintenance cycles.

Types of Regression Testing:
Corrective: Rerunning existing tests to check if the software still works as intended.

Retest-all: Re-executing all tests in the existing test suite.

Selective: Running a subset of tests that are impacted by the recent code changes.

Progressive: Developing new test cases for changed features while rerunning old test cases.

Complete: Comprehensive testing of the entire system to ensure all functionalities work correctly.

Tools for Regression Testing:
Automated Tools: Selenium, JUnit, TestNG, QTP, and others are commonly used to automate regression testing due to the repetitive nature of the tests.

Continuous Integration Tools: Jenkins, Travis CI, CircleCI, and others integrate regression testing into the build process, enabling automatic testing after every code commit.

Best Practices:
Automate Regression Tests: Automation saves time and resources, especially for large test suites.

Prioritize Test Cases: Focus on areas most likely to be affected by changes, such as critical paths and high-risk areas.

Maintain Test Suite: Regularly update and clean up the test suite to remove obsolete or redundant tests.

Use Version Control: Keep track of changes in the test suite using version control systems.

Integrate with CI/CD Pipelines: Ensure that regression tests are part of the continuous integration and continuous delivery (CI/CD) pipelines to catch issues early.

Challenges:

Time-Consuming: Running a full regression test suite can be time-consuming, especially for large applications.

Resource-Intensive: Requires significant computational resources, particularly for automated testing.

Maintenance: Keeping the test suite up to date with the latest changes can be challenging.

Conclusion: Regression testing is a critical component of software maintenance that helps ensure software quality over time. By systematically re-testing the software after changes, developers and testers can ensure that new updates do not disrupt existing functionalities, thereby maintaining a stable and reliable product.

Q.4 What are stub and driver? **(AKTU 2021-22)**

Ans: In software testing, particularly in unit testing and integration testing, stubs and drivers are used to simulate the behavior of missing components, allowing testers to test individual parts of the software in isolation. Here's an explanation of what stubs and drivers are, along with their roles and differences:

Stub

Definition: A stub is a piece of code used to simulate the behavior of a lower-level component (dependency) that a module under test interacts with. Stubs provide predefined responses to calls made by the module being tested, allowing the tester to isolate and test the module independently of its lower-level dependencies.

Purpose:

To replace a called component or module that is not yet developed or is not available for testing.

To provide controlled responses to the calls made by the module under test.

To isolate the module under test from its dependencies, ensuring that tests focus solely on the module's behavior.

Usage:

In unit testing, when a module calls methods or functions of another module that is not yet implemented.

In integration testing, to simulate the behavior of external systems or components that are unavailable or difficult to access.

Driver

Definition: A driver is a piece of code used to simulate the behavior of a higher-level component that calls the module under test. Drivers provide the necessary inputs and control to the module being tested, allowing the tester to initiate and control the testing process.

Purpose:

To replace the calling component or module that is not yet developed or is not available for testing.

To provide inputs and control to the module under test.

To initiate the execution of the module under test, ensuring that tests can be performed on lower-level modules.

Usage:

In unit testing, when the module under test is called by higher-level components that are not yet implemented.

In integration testing, to simulate the behavior of client applications or higher-level systems that interact with the module under test.

Differences between Stubs and Drivers

Aspect	Stub	Driver
Purpose	Simulates lower-level modules or dependencies.	Simulates higher-level modules or callers.
Direction	Called by the module under test.	Calls the module under test.
Usage	Provides predefined responses to the module.	Provides inputs and controls the execution of the module.
Context	Used when lower-level modules are missing.	Used when higher-level modules are missing.
Example	Simulating a payment gateway service.	Simulating a test interface for a payment processor.

Conclusion

Stubs and drivers are essential tools in software testing that allow for effective isolation and testing of individual modules. By using stubs and drivers, testers can create controlled environments to thoroughly test each part of the software, even when some components are not yet developed or available. This approach helps in identifying and fixing issues early in the development cycle, leading to more robust and reliable software.

Q.5 How do you describe software interface? **(AKTU 2018-19)**

Ans: A software interface is a critical aspect of software design that defines how different components of a software system interact with each other and with external entities. It can be understood as a set of rules and protocols that dictate how software components should communicate and operate together. Here's a detailed description of a software interface, its types, and its significance:

Definition

A software interface is a shared boundary across which two or more separate components of a computer system exchange information. This can include the methods, data formats, and communication protocols that enable this exchange. Interfaces ensure that different software modules can work together, even if they are developed independently or at different times.

Types of Software Interfaces

User Interface (UI):

Description: The part of the software that interacts directly with the user. It includes graphical elements like buttons, menus, and text fields (Graphical User Interface or GUI), or command-line

prompts (Command-Line Interface or CLI).

Purpose: To provide an intuitive and efficient way for users to interact with the software.

Example: The dashboard of a web application, a mobile app screen, or a console window for a CLI tool.

Application Programming Interface (API):

Description: A set of functions and procedures that allow different software applications to communicate with each other. APIs define how software components should interact and what data they can exchange.

Purpose: To enable integration between different systems, allowing them to work together or share data.

Example: RESTful APIs used in web services, library APIs provided by a software development kit (SDK).

Hardware Interface:

Description: The means by which software interacts with hardware components. This includes drivers and protocols that manage communication between the operating system and hardware devices.

Purpose: To ensure that software can control and use hardware resources effectively.

Example: Device drivers for printers, USB devices, or graphics cards.

Software Interface:

Description: The interface between different software components within the same system. This includes function calls, method invocations, and data structures that components use to interact.

Purpose: To enable modular design, where components can be developed and maintained independently.

Example: Public methods in an object-oriented class, microservices communication within a distributed application.

Characteristics of a Good Software Interface

Clarity:

Clearly defined inputs and outputs.

Unambiguous documentation that explains how to use the interface.

Consistency:

Consistent naming conventions and parameter types.

Predictable behavior across different scenarios.

Simplicity:

Minimalistic design that avoids unnecessary complexity.

Focus on essential functionalities.

Robustness:

Ability to handle unexpected inputs gracefully.

Error handling and validation mechanisms.

Flexibility:

Extensible design that allows for future enhancements without breaking existing functionality.

Support for optional parameters and default values.

Security:

Measures to prevent unauthorized access and data breaches.

Input validation and authentication mechanisms.

Significance of Software Interfaces

Modularity: Interfaces enable the development of modular systems where components can be developed, tested, and maintained independently.

Reusability: Well-designed interfaces promote code reuse by allowing existing components to be used in new applications or contexts.

Interoperability: Interfaces facilitate communication between different systems and technologies, enhancing compatibility and integration.

Scalability: By defining clear interaction points, interfaces help systems scale more easily as new components can be added without significant redesign.

Maintainability: With clear and consistent interfaces, maintaining and updating software becomes easier, reducing the likelihood of errors during changes.

Q.6 Compare & contrast between black-box testing & white-box testing. **(AKTU 2021-22)**

Ans:

Black Box Testing	**White Box Testing**
It is a way of software testing in which the internal structure or the program or the code is hidden and nothing is known about it.	It is a way of testing the software in which the tester has knowledge about the internal structure or the code or the program of the software.
It is mostly done by software testers.	It is mostly done by software developers.
No knowledge of implementation is needed.	Knowledge of implementation is required.
It can be referred as outer or external software testing.	It is the inner or the internal software testing.
It is functional test of the software.	It is structural test of the software.
This testing can be initiated on the basis of requirement specifications document.	This type of testing of software is started after detail design document.
No knowledge of programming is required.	It is mandatory to have knowledge of programming.
It is the behavior testing of the software.	It is the logic testing of the software.
It is applicable to the higher levels of testing of software.	It is generally applicable to the lower levels of software testing.
It is also called closed testing.	It is also called as clear box testing.
It is least time consuming.	It is most time consuming.
It is not suitable or preferred for algorithm testing.	It is suitable for algorithm testing.
Can be done by trial and error ways and methods.	Data domains along with inner or internal boundaries can be better tested.
Example: search something on google by using keywords	**Example:** by input to check and verify loops

Q.7 What are the various objectives of software testing? **(AKTU 2021-22)**

Ans: Software testing is a critical phase in the software development lifecycle (SDLC), aimed at ensuring that the software meets its requirements and functions as expected. The objectives of software testing can be summarized as follows:

1. Verification of Requirements

Objective: Ensure that the software meets all specified requirements.

Details: This involves checking whether the software correctly implements the functionalities and features as described in the requirement specifications.

2. Validation of Functionality

Objective: Confirm that the software behaves as expected under various conditions.

Details: This includes functional testing to verify that all features work correctly, both individually (unit testing) and collectively (integration testing).

3. Detection of Defects

Objective: Identify and fix defects (bugs) in the software.

Details: Testing helps uncover errors in the software code that could lead to incorrect behavior, crashes, or other issues.

4. Ensuring Quality

Objective: Assess the overall quality of the software.

Details: This includes evaluating various quality attributes such as reliability, usability, performance, and maintainability.

5. Providing Confidence

Objective: Build confidence in the software's reliability and performance.

Details: Thorough testing reassures stakeholders that the software is robust and dependable.

6. Preventing Future Issues

Objective: Proactively identify potential issues before they occur in the production environment.

Details: By identifying and addressing potential problems early, testing helps prevent costly fixes and downtime in the future.

7. Ensuring Compliance

Objective: Verify that the software complies with industry standards and regulations.

Details: This is particularly important in regulated industries such as healthcare, finance, and aerospace.

8. Improving User Experience

Objective: Ensure the software is user-friendly and meets user expectations.

Details: Usability testing focuses on the ease of use and overall experience for the end-users, ensuring the software is intuitive and pleasant to use.

9. Validating Performance

Objective: Ensure the software performs well under expected and peak conditions.

Details: Performance testing, including load testing, stress testing, and scalability testing, assesses how the software handles varying levels of user activity.

10. Ensuring Security

Objective: Identify and address security vulnerabilities.

Details: Security testing involves checking for potential security threats, including vulnerabilities to attacks, data breaches, and other malicious activities.

11. Facilitating Maintenance

Objective: Make future maintenance easier and more efficient.

Details: By ensuring the software is well-tested and free of defects, the need for extensive future maintenance is reduced, and updates can be made more smoothly.

12. Assessing Compatibility

Objective: Ensure the software is compatible with different environments.

Details: Compatibility testing checks how well the software works across various hardware, operating systems, network environments, and browsers.

13. Ensuring Accurate Documentation

Objective: Verify that all user manuals, help files, and other documentation are accurate and helpful.

Details: This ensures that end-users and maintenance teams have the correct information to use and support the software.

Conclusion

The objectives of software testing encompass a broad range of goals, all aimed at ensuring that the software is reliable, functional, and user-friendly. By systematically identifying and addressing issues, testing helps create high-quality software that meets user needs and performs reliably in real-world conditions.

Q.8 Explain about the structure chart and all its types with suitable example. **(AKTU 2022-23)**

Ans: Structure Chart represents the hierarchical structure of modules. It breaks down the entire system into the lowest functional modules and describes the functions and sub-functions of each module of a system in greater detail. This article focuses on discussing Structure Charts in detail.

What is a Structure Chart?

Structure Chart partitions the system into black boxes (functionality of the system is known to the users, but inner details are unknown).

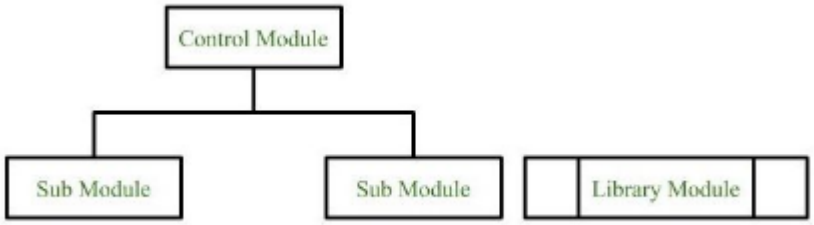
1. Inputs are given to the black boxes and appropriate outputs are generated.
2. Modules at the top level are called modules at low level.
3. Components are read from top to bottom and left to right.
4. When a module calls another, it views the called module as a black box, passing the required parameters and receiving results.

Symbols in Structured Chart

1. Module

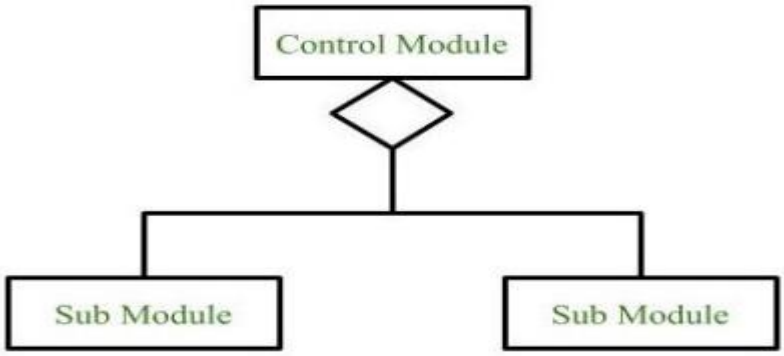
It represents the process or task of the system. It is of three types:

- **Control Module**: A control module branches to more than one submodule.
- **Sub Module:** Sub Module is a module which is the part (Child) of another module.
- **Library Module:** Library Module are reusable and invokable from any module.



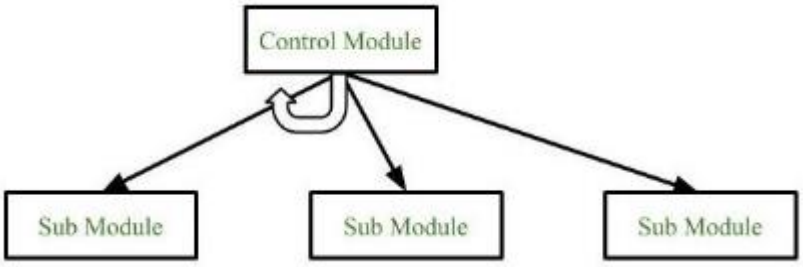
2. Conditional Call

It represents that control module can select any of the sub module on the basis of some condition.



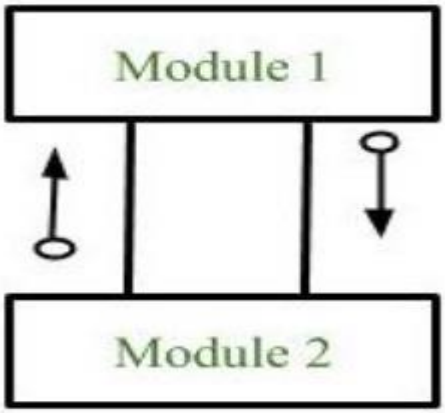
3. Loop (Repetitive call of module)

It represents the repetitive execution of module by the sub module. A curved arrow represents a loop in the module. All the submodules cover by the loop repeat execution of module.



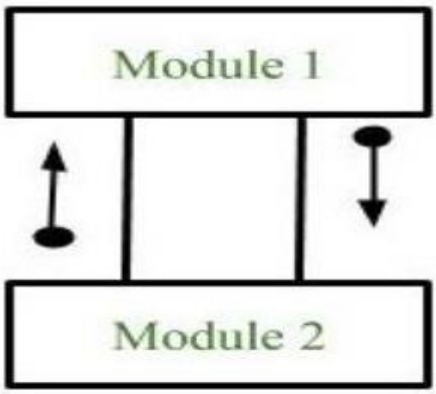
4. Data Flow

It represents the flow of data between the modules. It is represented by a directed arrow with an empty circle at the end.



5. Control Flow

It represents the flow of control between the modules. It is represented by a directed arrow with a filled circle at the end.

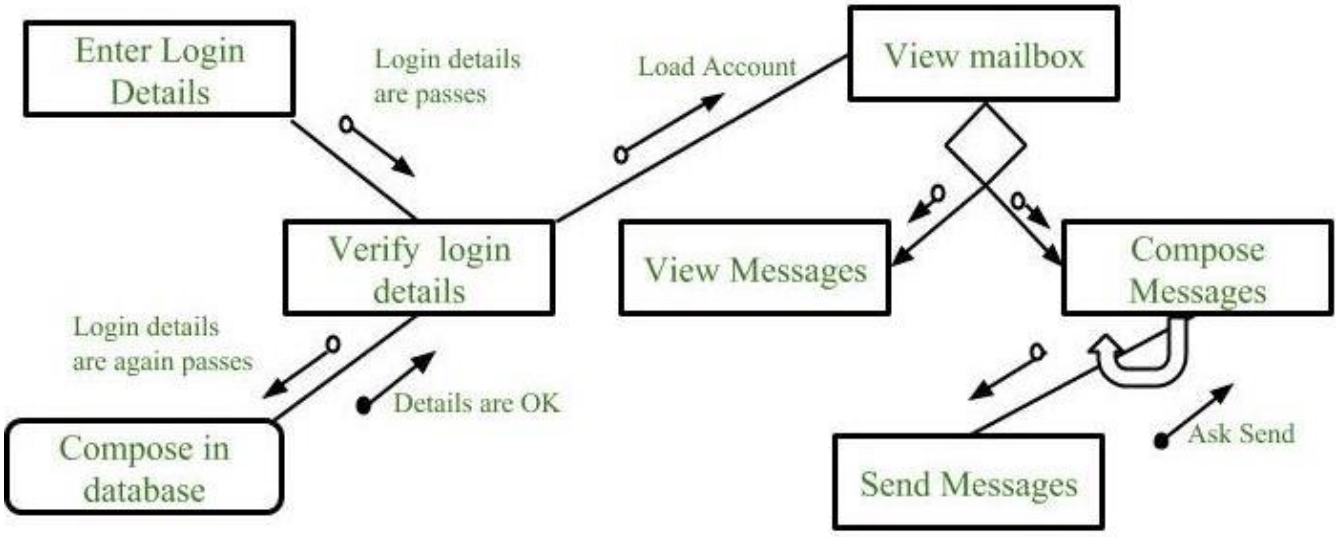


6. Physical Storage

It is that where all the information are to be stored.

Example

Structure chart for an Email server



Types of Structure Chart

1. **Transform Centered Structure:** This type of structure chart are designed for the systems that receives an input which is transformed by a sequence of operations being carried out by one module.
2. **Transaction Centered Structure:** These structures describe a system that processes a number of different types of transaction.

Q.9 Differentiate between alpha & beta testing. **(AKTU 2022-23)**

Ans: Alpha and beta testing are both crucial stages in the software testing process, but they occur at different points and serve distinct purposes. Here's a comparison between alpha and beta testing:

Alpha Testing:

Definition:

Alpha testing is conducted by the development team, usually in a controlled environment, before the software is released to external users.

Objective:

The primary goal of alpha testing is to identify defects and issues within the software before it undergoes further testing and is released to a wider audience.

Scope:

Alpha testing typically focuses on validating the functionality, usability, and overall quality of the software from an internal perspective.

Environment:

Alpha testing is conducted in a controlled environment, often within the development organization's premises, using simulated or representative data.

Participants:

Alpha testing involves testers from within the development team, including developers, testers, and other stakeholders.

Feedback:

Feedback from alpha testing is used to improve the software's quality, address issues, and make necessary adjustments before releasing it for broader testing.

Beta Testing:

Definition:

Beta testing is conducted by a select group of external users, often referred to as beta testers, in a real-world environment, before the software's official release.

Objective:

The main objective of beta testing is to gather feedback from actual users in diverse environments to identify any remaining issues, assess usability, and gather suggestions for improvement.

Scope:

Beta testing aims to validate the software's functionality, usability, performance, and compatibility in real-world scenarios, across different hardware, software, and network configurations.

Environment:

Beta testing is conducted in a real-world environment, typically on the users' own devices and networks, using actual data and usage patterns.

Participants:

Beta testing involves a selected group of external users who represent the software's target

audience. These users voluntarily participate in testing and provide feedback.

Feedback:

Feedback from beta testing helps the development team identify any remaining issues, assess user satisfaction, and make final refinements before the software's official release.

Key Differences:

Timing:

Alpha testing occurs before beta testing and is conducted by the development team.

Beta testing occurs after alpha testing and involves external users testing the software in a real-world environment.

Participants:

Alpha testing involves testers from within the development organization.

Beta testing involves external users who represent the software's target audience.

Environment:

Alpha testing is conducted in a controlled environment, often using simulated data.

Beta testing is conducted in a real-world environment, using actual data and user scenarios.

Purpose:

Alpha testing aims to identify defects and issues within the software before release.

Beta testing aims to gather feedback from real users to assess usability, performance, and overall satisfaction.

Scope:

Alpha testing focuses on internal validation of the software's functionality and quality.

Beta testing focuses on external validation of the software's usability and effectiveness in real-world scenarios.

In summary, while alpha testing is an internal process aimed at identifying defects and issues before release, beta testing involves external users testing the software in real-world conditions to gather feedback and assess usability. Both stages are crucial for ensuring the software meets user expectations and quality standards before its official release.

Q.10 Write short notes on: **(AKTU 2022-23)**

(a) Walkthrough

(b) Code Inspection

Ans: Walkthrough:

Walkthrough is a method of conducting informal group/individual review. In a walkthrough, author describes and explain work product in a informal meeting to his peers or supervisor to get feedback. Here, validity of the proposed solution for work product is checked.

It is cheaper to make changes when design is on the paper rather than at time of conversion. Walkthrough is a static method of quality assurance. Walkthrough are informal meetings but with purpose.

Inspection:

An inspection is defined as formal, rigorous, in depth group review designed to identify problems as close to their point of origin as possible. Inspections improve reliability, availability, and maintainability of software product.

Anything readable that is produced during the software development can be inspected. Inspections can be combined with structured, systematic testing to provide a powerful tool for creating defect-free programs.

Inspection activity follows a specified process and participants play well-defined roles. An inspection team consists of three to eight members who plays roles of moderator, author, reader, recorder and inspector.

For example, designer can acts as inspector during code inspections while a quality assurance representative can act as standard enforcer.

Stages in the inspections process:

- **Planning:** Inspection is planned by moderator.
- **Overview meeting:** Author describes background of work product.
- **Preparation:** Each inspector examines work product to identify possible defects.
- **Inspection meeting:** During this meeting, reader reads through work product, part by part and inspectors points out the defects for every part.
- **Rework:** Author makes changes to work product according to action plans from the inspection meeting.
- **Follow-up :** Changes made by author are checked to make sure that everything is correct

Q.11 What is Integration Testing? Explain different approaches used for integration testing**.**

(AKTU 2023-24)

Ans. Integration Testing:

Integration Testing is a type of software testing in which individual components or modules of a software system are combined and tested as a group. The goal of integration testing is to identify issues that arise when different modules or systems interact with each other, ensuring that they work together as expected.

This level of testing occurs after unit testing (where individual modules are tested in isolation) and before system testing (where the entire system is tested). Integration testing verifies that the interfaces between modules, the data flow between them, and their interaction with other systems work correctly.

Objectives of Integration Testing:

1. **Verify Communication**: Ensure that the different modules or components of the system communicate correctly with each other.
2. **Detect Interface Issues**: Identify problems such as mismatched data formats, incorrect parameters, or issues with data passing between modules.
3. **Check Functional Behavior**: Verify that the integrated modules function as intended when working together.
4. **Ensure Performance**: Test how the combined modules handle the expected workload.

Different Approaches to Integration Testing:

There are several approaches to integration testing, and the choice of approach depends on the system

architecture, project requirements, and development methodology. The most common approaches include:

1. Top-Down Integration Testing

- **Description**: In this approach, integration starts from the top-level modules (usually the main control module) and proceeds downwards towards the lower-level modules.
- **How it works**: Higher-level modules are tested first, and lower-level modules are simulated using **stubs** (dummy modules that mimic the behavior of missing components). As the testing progresses, the lower-level modules are added to the system.
- **Advantages**:
  - Early testing of high-level functionality.
  - Helps identify architectural flaws early.
- **Disadvantages**:
  - Lower-level modules may not be fully tested until later stages.
  - Stubs can lead to incomplete testing of lower-level modules.

2. Bottom-Up Integration Testing

- **Description**: This approach starts testing from the lower-level modules and works upwards to the higher-level modules.
- **How it works**: The lowest-level modules are tested first, and higher-level modules are gradually integrated. **Drivers** (dummy modules that simulate higher-level modules) are used to simulate the interaction with missing modules.
- **Advantages**:
  - Thorough testing of low-level modules and their interactions.
  - Lower-level components are fully tested before they are integrated.
- **Disadvantages**:
  - Higher-level functionality isn't tested until later in the process.
  - Requires more time to complete testing of high-level modules.

3. Big Bang Integration Testing

- **Description**: In this approach, all modules or components are integrated together at once and then tested as a whole system.
- **How it works**: All components are integrated in one step, and the system is tested in its entirety, usually using real modules.
- **Advantages**:
  - Simple and fast to start testing.
  - No need to create stubs or drivers.
- **Disadvantages**:
  - Difficult to isolate and diagnose issues because everything is tested at once.
  - High risk of finding numerous errors at once, which can make debugging complex.

4. Incremental Integration Testing

- **Description**: In this approach, modules are integrated and tested incrementally. The modules are added one at a time or in small groups, and testing is done after each integration.

- **How it works**: A single or small group of modules is integrated and tested before more modules are added. The process continues until all components are integrated.
- **Advantages**:
  - Easier to isolate defects as modules are tested in smaller groups.
  - Allows testing of functionality step-by-step.
- **Disadvantages**:
  - Takes more time compared to Big Bang testing, as testing is done incrementally.

5. Sandwich Integration Testing (Hybrid Approach)

- **Description**: This is a combination of both **Top-Down** and **Bottom-Up** integration approaches. It combines the strengths of both strategies.
- **How it works**: Testing begins at both the top and bottom levels simultaneously, with integration proceeding towards the middle layers. Stubs and drivers are used as needed.
- **Advantages**:
  - Combines the advantages of both Top-Down and Bottom-Up testing.
  - Allows for parallel development of both higher and lower-level modules.
- **Disadvantages**:
  - Can be complex to manage and implement.
  - Requires a thorough understanding of the system's architecture.

6. Continuous Integration Testing

- **Description**: In modern agile development environments, continuous integration (CI) involves frequent integration of code changes and automated testing to ensure that the system continues to function correctly as new changes are introduced.
- **How it works**: Developers frequently integrate their individual modules or changes into a shared repository, where automated integration tests are run continuously (usually after each code commit).
- **Advantages**:
  - Helps identify integration issues early and often.
  - Ensures the system is always in a potentially shippable state.
- **Disadvantages**:
  - Relies on robust and efficient automation.
  - Requires a mature CI/CD pipeline and good test coverage.

BUDDHA SERIES

(Unit Wise Solved Question & Answers)

Course – B.Tech. (CSE)

College – Buddha Institute of Technology

(AKTU CODE-525)

Department: Computer Science & Engineering(AIML/DS)

Subject: Software Engineering (BCS 601)

Faculty Name: Satish Kumar

Unit - 5

Q.1 List the points of differences between software Re-engineering and Reverse engineering. **(AKTU 2021-22)**

Ans:

Forward Engineering	Reverse Engineering
In forward engineering, the application are developed with the given requirements.	In reverse engineering or backward engineering, the information are collected from the given application.
Forward Engineering is a high proficiency skill.	Reverse Engineering or backward engineering is a low proficiency skill.
Forward Engineering takes more time to develop an application.	While Reverse Engineering or backward engineering takes less time to develop an application.
The nature of forward engineering is Prescriptive.	The nature of reverse engineering or backward engineering is Adaptive.
In forward engineering, production is started with given requirements.	In reverse engineering, production is started by taking the existing products.
The example of forward engineering is the construction of electronic kit, construction of DC MOTOR , etc.	An example of backward engineering is research on Instruments etc.
Forward engineering Starts with requirements analysis and design, then proceeds to implementation and testing.	Reverse engineering Starts with an existing software system and works backward to understand its structure, design, and requirements.

Q.2 Explain Risk management in detail. Also discuss the points that differentiate project risk from technical risk. **(AKTU 2021-22)**

Ans: Risk management is a systematic process of identifying, assessing, and prioritizing risks, followed by coordinated efforts to minimize, monitor, and control the probability and/or impact of unfortunate events. The goal of risk management is to ensure that an organization achieves its objectives and minimizes adverse effects from various types of risks.

Key Steps in Risk Management

Risk Identification:

Process: Identifying potential risks that could affect the project or organization.

Tools: Brainstorming sessions, SWOT analysis (Strengths, Weaknesses, Opportunities, Threats), checklists, and expert judgment.

Risk Assessment:

Qualitative Analysis: Assessing the impact and likelihood of identified risks using a relative scale (e.g., high, medium, low).

Quantitative Analysis: Using numerical methods to evaluate the potential impact of risks. Techniques include Monte Carlo simulation, decision tree analysis, and sensitivity analysis.

Risk Prioritization:

Process: Ranking risks based on their potential impact and likelihood to determine which risks require immediate attention.

Tools: Risk matrix (impact vs. likelihood), Pareto analysis.

Risk Response Planning:

Strategies:

Avoidance: Changing plans to circumvent the risk.

Mitigation: Reducing the probability or impact of the risk.

Transfer: Shifting the risk to a third party (e.g., insurance).

Acceptance: Acknowledging the risk and preparing to manage its impact.

Tools: Contingency planning, insurance, contracts.

Risk Monitoring and Control:

Process: Tracking identified risks, monitoring residual risks, identifying new risks, and evaluating the effectiveness of risk response strategies.

Tools: Risk audits, periodic risk reviews, status meetings, and variance analysis.

Communication and Reporting:

Process: Keeping stakeholders informed about risk management activities, status, and changes.

Tools: Risk registers, risk reports, dashboards.

Differentiating Project Risk from Technical Risk

While project risk and technical risk are often interrelated, they focus on different aspects of a project and have distinct characteristics.

Project Risk

Scope:

Broad and includes all aspects of the project (e.g., schedule, budget, resources, stakeholders).

Encompasses external and internal factors that could affect project success.

Examples:

Budget overruns.

Schedule delays.

Resource shortages.

Stakeholder conflicts.

Regulatory changes.

Management Focus:

Emphasis on overall project planning and execution.

Risk management strategies involve high-level project decisions, such as scope changes or resource reallocation.

Technical Risk

Scope:

Specific to the technical aspects and deliverables of the project.

Focuses on the technology, processes, and methodologies used in the project.

Examples:

Technical feasibility issues.

Performance failures or deficiencies.

Integration problems.

Technological obsolescence.

Software bugs or hardware failures.

Management Focus:

Emphasis on the technical integrity and performance of the project deliverables.

Risk management strategies involve technical solutions, such as prototyping, testing, and adopting new technologies or methodologies.

Comparison of Project Risk and Technical Risk

Aspect	Project Risk	Technical Risk
Scope	Broad, encompassing all project areas	Narrow, focusing on technical deliverables
Examples	Budget overruns, schedule delays, stakeholder issues	Technical failures, integration issues, performance gaps
Impact	Affects project success, delivery, and stakeholder satisfaction	Affects technical performance, quality, and functionality
Management Focus	Project planning, resource allocation, stakeholder management	Technical solutions, testing, and technology selection
Risk Owners	Project managers, stakeholders, senior management	Engineers, technical leads, IT specialists
Mitigation Strategies	Scope changes, contingency funds, resource adjustments	Prototyping, additional testing, technology upgrades

In summary, effective risk management requires a comprehensive understanding of both project and technical risks, tailored strategies to address each type, and continuous monitoring to adapt to new challenges and ensure project success.

Q.3 What is cost analysis in context of software? Explain COCOMO model with the help of schematic diagram. **(AKTU 2021-22)**

Ans: Cost Benefit analysis is thing that everyone must do so as to think of a powerful or an efficient system. But while thinking out on cost and benefit analysis, we also need to find out factors that really affect benefits and costs of system. In developing cost estimates for a system, we need to consider some of cost elements. Some elements among them are hardware, personnel, facility, operating and supply cost.

COCOMO Model

Boehm proposed COCOMO (Constructive Cost Estimation Model) in 1981.COCOMO is one of the most generally used software estimation models in the world. COCOMO predicts the efforts and schedule of a software product based on the size of the software.

The necessary steps in this model are:

1. Get an initial estimate of the development effort from evaluation of thousands of delivered lines of source code (KDLOC).
2. Determine a set of 15 multiplying factors from various attributes of the project.
3. Calculate the effort estimate by multiplying the initial estimate with all the multiplying factors i.e., multiply the values in step1 and step2.

The initial estimate (also called nominal estimate) is determined by an equation of the form used in the static single variable models, using KDLOC as the measure of the size. To determine the initial effort E_i in person-months the equation used is of the type is shown below

$$\mathbf{E_i=a*(KDLOC)b}$$

The value of the constant a and b are depends on the project type.

According to Boehm, software cost estimation should be done through three stages:

1. Basic Model
2. Intermediate Model
3. Detailed Model

1. Basic COCOMO Model: The basic COCOMO model provide an accurate size of the project parameters. The following expressions give the basic COCOMO estimation model:

$$\mathbf{Effort=a_1*(KLOC)a_2\ PM}$$

$$\mathbf{Tdev=b_1*(efforts)b_2\ Months}$$

Where, **KLOC** is the estimated size of the software product indicate in Kilo Lines of Code,

a_1,a_2,b_1,b_2 are constants for each group of software products,

Tdev is the estimated time to develop the software, expressed in months,

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

Estimation of development effort

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

Organic: Effort = 2.4(KLOC) 1.05 PM

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

Embedded: Effort = 3.6(KLOC) 1.20 PM

Estimation of development time

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

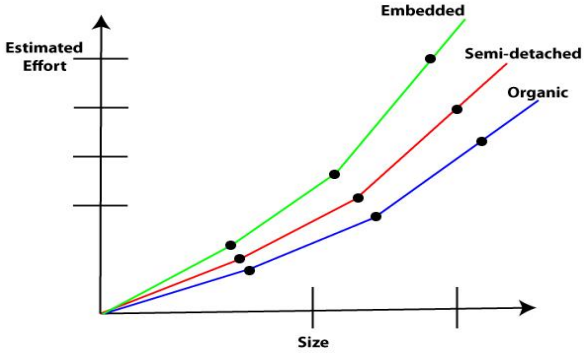
Organic: Tdev = 2.5(Effort) 0.38 Months

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

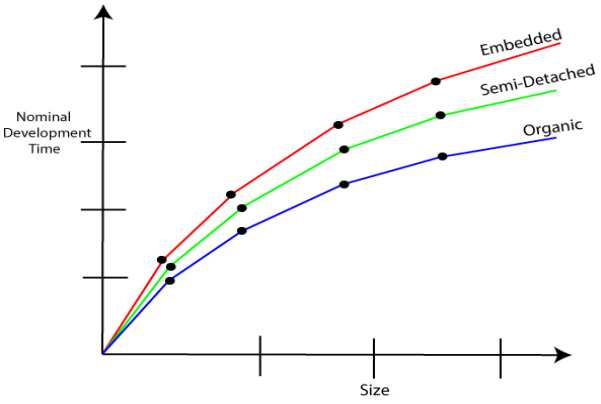
Embedded: Tdev = 2.5(Effort) 0.32 Months

Some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes. Fig shows a plot of estimated effort versus product size. From fig, we can observe that the effort is somewhat superliner in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size.

The development time versus the product size in KLOC is plotted in fig. From fig it can be observed that the development time is a sub linear function of the size of the product, i.e. when the size of the product increases by two times, the time to develop the product does not double but rises moderately. This can be explained by the fact that for larger products, a larger number of activities which can be carried out concurrently can be identified. The parallel activities can be carried out simultaneously by the engineers. This reduces the time to complete the project. Further, from fig, it can be observed that the development time is roughly the same for all three categories of products. For example, a 60 KLOC program can be developed in approximately 18 months, regardless of whether it is of organic, semidetached, or embedded type.



Effort versus product size



Development time versus size

Q.4 Differentiate between adaptive and corrective maintenance? **(AKTU 2021-22)**

Ans:

Corrective Maintenance	Preventive Maintenance
CM is performed after asset failure or after anything goes wrong.	PM is performed before asset failure or before anything goes wrong.
It is less complex and simple process	It is more complex as compared to CM

Corrective Maintenance	Preventive Maintenance
because it does not involve any planning to prevent asset from failure.	because t involves planning to prevent assets from failure.
It can be more expensive than PM as some equipment failure causes greater damage to system.	It can be expensive but it prevents assets from failure.
It can affect overall system as some assets failure can cause greater loss in production.	It prevents loss in production by reducing chances of failure occurrence.
CM leads to expensive repairs along with unscheduled repairs.	PM mainly aims to avoid expensive repairs and corrective actions.
It is performed at random intervals as it is only performed when a failure occurs.	It is performed at regular intervals as asset maintenance is important and should be checked regularly to avoid any failure occurrence.
CM generally increases need for assets preventive actions.	PM generally decreases need for corrective actions.
CM overall decreases lifecycle of assets.	PM overall increases life cycle of assets.
This process results in loss of production, loss of product quality, loss of time.	This process leads to increase in production, increase in product quality, no loss of production time.
It is not better for safety of employees and working environment as it increases risk of injury.	It is better for safety of employees and working environment as it reduces risk of injury.
Downtime of assets is more in CM.	Downtime of assets is less in PM.
It requires a greater number of employees or technicians to perform CM.	It requires a smaller number of employees or technicians to perform PM.
It increases workload for technicians.	It decreases workload for technicians.

Q.5 Why is Software maintenance required? Explain types of maintenances with examples.

(AKTU 2021-22)

Ans: Software maintenance refers to the process of modifying and updating a software system after it has been delivered to the customer. This can include fixing bugs, adding new features, improving performance, or updating the software to work with new hardware or software systems. The goal of software maintenance is to keep the software system working correctly, efficiently, and securely, and to ensure that it continues to meet the needs of the users.

There are several key aspects of software maintenance, including:

1. Bug fixing: The process of finding and fixing errors and problems in the software.
2. Enhancements: The process of adding new features or improving existing features to meet the evolving needs of the users.
3. Performance optimization: The process of improving the speed, efficiency, and reliability of the software.
4. Porting and migration: The process of adapting the software to run on new hardware or software platforms.
5. Re-engineering: The process of improving the design and architecture of the software to make it more maintainable and scalable.
6. Documentation: The process of creating, updating, and maintaining the documentation for the software, including user manuals, technical specifications, and design documents.

Software maintenance is a critical part of the software development life cycle and is necessary to ensure that the software continues to meet the needs of the users over time. It is also important to consider the cost and effort required for software maintenance when planning and developing a software system.

Software maintenance is the process of modifying a software system after it has been delivered to the customer. The goal of maintenance is to improve the system's functionality, performance, and reliability and to adapt it to changing requirements and environments.

There are several types of software maintenance, including:

- **Corrective maintenance:** This involves fixing errors and bugs in the software system.

Patching: It is a emergency fixes implemented mainly due to pressure from management. Patching is done for corrective maintenance but it gives rise to unforeseen future-errors due to lack of proper impact analysis.

- **Adaptive maintenance:** This involves modifying the software system to adapt it to changes in the environment, such as changes in hardware or software, government polices, business rules.
- **Perfective maintenance**: This involves improving the functionality, performance, reliability and restructuring the software system to improve the changeability.
- **Preventive maintenance:** This involves taking measures to prevent future problems, such as optimization, updating documentation, reviewing and testing the system, and implementing preventive measures such as backups.

Software maintenance is a continuous process that occurs throughout the entire life cycle of the software system. It is important to have a well-defined maintenance process in place, which includes testing and validation, version control, and communication with stakeholders

Q.6 Define corrective & preventive maintenance. **(AKTU 2022-23)**

Ans:

Corrective Maintenance	**Preventive Maintenance**
CM is performed after asset failure or after anything goes wrong.	PM is performed before asset failure or before anything goes wrong.
It is less complex and simple process because it does not involve any planning to prevent asset from failure.	It is more complex as compared to CM because involves planning to prevent assets from failure.
It can be more expensive than PM as some equipment failure causes greater damage to system.	It can be expensive but it prevents assets from failure.
It can affect overall system as some assets failure can cause greater loss in production.	It prevents loss in production by reducing chances of failure occurrence.
CM leads to expensive repairs along with unscheduled repairs.	PM mainly aims to avoid expensive repairs and corrective actions.

Q.7 Write short notes on: **(AKTU 2022-23)**

(i) Function Point

(ii)Defect, Fault, Failure

Ans: (i) Function Points

Function Points (FP) are a standardized unit of measure used to express the amount of business functionality an information system provides to a user. Here is a sorted and organized collection of notes on Function Points:

Overview

Definition: Function Points measure software size by quantifying the functionality provided to the user based on the user's external view of the system.

Purpose: Used to estimate the cost, effort, and time required for software development and maintenance.

Components of Function Points

Inputs (External Inputs - EI):
Data or control information that comes from outside the system and triggers an internal process.
Examples: User entries, data from other systems.
Outputs (External Outputs - EO):
Data or control information that exits the system, produced by internal processing.
Examples: Reports, error messages.
Inquiries (External Inquiries - EQ):
User-initiated requests that require immediate response but do not involve significant processing or alteration of data.
Examples: Online queries, prompts.
Internal Logical Files (ILF):
Logical files maintained within the system boundary that store data for the system's use.
Examples: Databases, internal data tables.
External Interface Files (EIF):
Logical files maintained outside the system boundary but are used or referenced by the system.
Examples: Shared files, external databases.
Calculation of Function Points
Identify Function Types:
List all inputs, outputs, inquiries, internal logical files, and external interface files.
Assign Complexity Weightings:
Each function type is assigned a complexity level (low, average, high) based on predefined criteria.
Each complexity level corresponds to a weight or point value (e.g., low = 3, average = 4, high = 6 for inputs).
Calculate Unadjusted Function Points (UFP):
Multiply the number of functions by their respective weights and sum up the values.
Formula: UFP=∑(Number of functions×Weight)UFP=∑(Number of functions×Weight).
Adjust for Environmental Factors:
Consider the impact of various technical and operational factors (such as performance, complexity, usability).
Use the Value Adjustment Factor (VAF) which typically ranges from 0.65 to 1.35.
Formula: Adjusted Function Points=UFP×VAFAdjusted Function Points=UFP×VAF.
Uses and Benefits
Estimation: Helps in estimating project size, effort, cost, and duration.
Productivity Measurement: Provides a basis for comparing productivity across projects and teams.
Benchmarking: Facilitates industry-wide comparisons and benchmarking of software projects.
Project Management: Aids in tracking project progress and managing scope changes.
Advantages
Standardization: Provides a consistent method for measuring software size.
User-oriented: Focuses on the functionality required by the user.

Language-agnostic: Applicable regardless of the programming language or technology used.

Challenges and Limitations

Subjectivity: Some degree of subjectivity in determining complexity levels and weights.

Initial Learning Curve: Requires training and experience to apply accurately.

Maintenance Effort: Regular updates needed to maintain the relevance of Function Point metrics as systems evolve.

Common Uses in Industry

Software Development: Used by developers and project managers to estimate effort and track progress.

Costing and Budgeting: Utilized by financial planners to budget for software projects.

Vendor Contracts: Employed in outsourcing contracts to define deliverables and measure vendor performance.

Conclusion

Function Points provide a valuable method for quantifying software functionality, aiding in estimation, productivity measurement, and project management. Despite some challenges, their benefits in standardization and user-orientation make them a widely adopted tool in the software industry.

(ii)

Defect

Definition

A defect is an imperfection or deficiency in a software product where it does not meet its requirements or specifications.

Also known as a bug, a defect is a flaw in the software that can cause it to behave unexpectedly or produce incorrect results.

Characteristics

Introduced During Development: Typically occurs during coding, design, or requirement gathering phases.

Detected in Testing: Often identified during testing phases by quality assurance teams.

Classifications:

Syntax Defects: Errors in the code syntax.

Logic Defects: Errors in the logic that lead to incorrect behavior or results.

Interface Defects: Issues in the interaction between different modules or systems.

Performance Defects: Issues that affect the performance of the software.

Examples

Incorrect calculation in a financial application.

Misspelled text in a user interface.

Failure to handle null values in input fields.

Fault

Definition

A fault is the actual cause of a defect in the software. It is the underlying issue in the code or design that leads to a defect.

A fault is sometimes referred to as an error.

Characteristics

Root Cause: Represents the root cause that leads to a defect.

Latent Faults: Faults that exist in the system but have not yet been detected or manifested as a defect.

Correction Required: Needs to be corrected to fix the defect and prevent it from causing a failure.

Examples

A missing semicolon in a line of code (syntax fault).

An incorrect algorithm implementation (logic fault).

An incorrect API call leading to improper data handling (interface fault).

Failure

Definition

A failure occurs when the software does not perform its intended function or deviates from expected behavior during execution.

It is the manifestation of a defect or fault in the operational environment.

Characteristics

Observable: Failures are observable and can be experienced by end-users or during testing.

Impact: Can cause significant disruption, data loss, or security breaches.

Immediate Attention Needed: Often requires immediate attention to prevent further issues.

Examples

A banking application crashes when a user tries to transfer money.

A website displays incorrect data after a user submits a form.

A mobile app fails to load and exits unexpectedly.

Comparison

Aspect	Defect	Fault	Failure
Definition	Flaw in the software product	Root cause of a defect	Deviation from expected behavior during execution
Occurrence	Found during development/testing	Exists in code or design	Occurs in operational environment
Detectability	Detected during testing	Identified through debugging/analysis	Observable by end-users or testers
Impact	Can cause incorrect behavior	Leads to defects	Causes software to deviate from expected behavior
Examples	Incorrect calculation, UI typo	Missing semicolon, incorrect algorithm	Application crash, incorrect data display

Summary

Defect: The symptom observed in the software that deviates from expected results.

Fault: The underlying cause or error in the code or design that results in a defect.

Failure: The actual manifestation of a defect during the execution of the software, leading to incorrect or unexpected behavior.

Understanding the distinctions between defects, faults, and failures is crucial in software development and maintenance for effectively identifying, diagnosing, and resolving issues to ensure software quality and reliability.

Q.8 Write briefly on CASE Tools. How to estimate cost, effort and schedule/duration.

(AKTU 2018-19)

Ans: Computer-Aided Software Engineering (CASE) tools are software applications designed to assist in the software development process. They automate various tasks involved in software development, enhancing productivity and ensuring quality by providing support for different stages of the software lifecycle.

Types of CASE Tools

Upper CASE Tools:

Support the early stages of software development, such as requirements analysis and design.

Examples: Diagramming tools, modeling tools, requirement management tools.

Lower CASE Tools:

Aid in the later stages, including implementation, testing, and maintenance.

Examples: Code generators, test management tools, debugging tools.

Integrated CASE Tools:

Provide comprehensive support throughout the entire software development lifecycle by integrating upper and lower CASE tools.

Examples: Integrated Development Environments (IDEs), software suites like IBM Rational Rose.

Benefits

Increased Productivity: Automates repetitive tasks, reducing manual effort.

Improved Quality: Ensures consistency and adherence to standards.

Enhanced Collaboration: Facilitates communication among team members through shared models and documentation.

Better Project Management: Provides tools for tracking progress, managing requirements, and controlling changes.

Common Features

Diagramming and Modeling: Create visual representations of system architecture and design.

Code Generation: Automatically generate source code from models.

Documentation: Generate and maintain project documentation.

Testing Support: Automate testing processes and manage test cases.

Configuration Management: Track changes and manage versions of software artifacts.

Estimating Cost, Effort, and Schedule/Duration

Cost Estimation

Analogous Estimation:

Uses historical data from similar projects to estimate costs.

Pros: Quick and easy.

Cons: May lack accuracy if past projects are not comparable.

Parametric Estimation:

Uses mathematical models to estimate costs based on project parameters (e.g., lines of code, function points).

Pros: More accurate with reliable models.

Cons: Requires accurate data and model calibration.

Bottom-Up Estimation:

Estimates costs by aggregating the costs of individual tasks or components.

Pros: Detailed and accurate.

Cons: Time-consuming and complex.

Effort Estimation

Expert Judgment:

Relies on the experience and intuition of experts to estimate effort.

Pros: Quick and leverages expertise.

Cons: Subjective and may vary between experts.

Delphi Technique:

A group of experts provides estimates anonymously, and the results are averaged.

Pros: Reduces bias and leverages collective wisdom.

Cons: Requires multiple rounds and coordination.

Algorithmic Models:

Uses formulas to estimate effort based on project characteristics (e.g., COCOMO - Constructive Cost Model).

Pros: Consistent and repeatable.

Cons: May need calibration for specific contexts.

Schedule/Duration Estimation

Critical Path Method (CPM):

Identifies the longest path of dependent tasks and calculates the minimum project duration.

Pros: Highlights critical tasks and dependencies.

Cons: May be complex for large projects.

Program Evaluation and Review Technique (PERT):

Uses optimistic, pessimistic, and most likely estimates to calculate expected task durations.

Pros: Accounts for uncertainty and variability.

Cons: Requires detailed task analysis.

Gantt Charts:

Visual representation of the project schedule, showing start and end dates for tasks.

Pros: Easy to understand and communicate.

Cons: May not capture dependencies as effectively as network diagrams.

Combining Estimates

Use Multiple Methods:

Combine different estimation techniques to improve accuracy (e.g., use both analogous and parametric methods).

Review and Refine:

Continuously review estimates and refine them as more information becomes available.

Contingency Planning:

Include buffers or contingency reserves to account for uncertainties and risks.

By using a combination of CASE tools for automation and various estimation techniques, project managers can more effectively plan, execute, and control software development projects, ensuring better predictability and success rates.

Q.9 What are the need and category for maintenance in software maintenance?
(AKTU 2018-19)

Ans: Need for Software Maintenance

Software maintenance is essential for several reasons, ensuring that the software remains useful, secure, and efficient throughout its lifecycle. Here are the key needs for software maintenance:

Key Needs

Bug Fixing:

Correction of Errors: Identifying and fixing bugs and errors that were not discovered during the initial testing phase.

Security Patches: Addressing vulnerabilities to protect against security threats.

Adaptation:

Environmental Changes: Adapting the software to new hardware, operating systems, or other external systems that interact with the software.

Regulatory Changes: Ensuring compliance with new legal or regulatory requirements.

Enhancement:

New Features: Adding new functionalities to meet evolving user requirements.

Performance Improvements: Optimizing the software for better performance and efficiency.

Prevention:

Preventive Measures: Implementing changes to prevent potential future problems or to improve the maintainability of the software.

Code Refactoring: Improving code structure without changing its external behavior to make future maintenance easier.

User Support:

User Assistance: Providing ongoing support to users, including training, troubleshooting, and help desk services.

Categories of Software Maintenance

Software maintenance is generally categorized into four main types, each addressing different aspects of maintaining and improving software systems:

1. Corrective Maintenance

Purpose: To fix defects and errors discovered after the software has been deployed.

Scope: Includes bug fixes, error corrections, and resolution of faults that affect functionality or performance.

Examples: Fixing a calculation error, resolving a crash issue, patching security vulnerabilities.

2. Adaptive Maintenance

Purpose: To modify the software so it remains effective in a changing environment.

Scope: Includes updates to ensure compatibility with new operating systems, hardware, or software interfaces.

Examples: Updating software to work with a new OS version, modifying an application to support new hardware devices.

3. Perfective Maintenance

Purpose: To enhance the software by adding new features or improving existing functionalities based on user feedback or changing requirements.

Scope: Includes performance improvements, user interface enhancements, and feature additions.

Examples: Adding a new reporting feature, improving load times, enhancing the user interface.

4. Preventive Maintenance

Purpose: To prevent future problems by addressing potential issues before they become actual defects.

Scope: Includes code optimization, refactoring, and updating documentation to improve software maintainability.

Examples: Refactoring code to reduce complexity, updating outdated documentation, conducting regular audits to detect potential issues.

Summary

Category	Purpose	Scope	Examples
Corrective	Fix defects and errors	Bug fixes, error corrections, resolving faults	Fixing a calculation error, patching security vulnerabilities
Adaptive	Adapt to changing environments	Updates for compatibility with new OS, hardware, or software interfaces	Updating for new OS, supporting new hardware devices
Perfective	Enhance functionality	Performance improvements, user interface enhancements, adding new features	Adding reporting feature, improving load times
Preventive	Prevent future issues	Code optimization, refactoring, updating documentation	Refactoring code, updating documentation, regular audits

Conclusion

Software maintenance is critical to ensure the longevity, security, and efficiency of software systems. By addressing various needs through corrective, adaptive, perfective, and preventive maintenance, organizations can keep their software relevant and functional, meeting both current and future demands.

Q.10 Discuss Software Configuration Management and various tasks in SCM process. Explain version control and various types of project risks. **(AKTU 2022-23)**

Ans: Software Configuration Management (SCM)

Overview

Software Configuration Management (SCM) is a discipline within software engineering that involves the systematic control of changes to the configuration and source code of a software system. Its primary goal is to ensure the integrity and traceability of the configuration throughout the software lifecycle.

Objectives of SCM

Version Control: Managing changes to software artifacts and maintaining different versions.

Change Control: Ensuring that all changes are systematically managed and documented.

Configuration Identification: Identifying the attributes and components of the software to control and track changes.

Configuration Auditing: Ensuring that configurations conform to their specified requirements.

Configuration Status Accounting: Recording and reporting the status of configuration items and change requests.

Tasks in the SCM Process

Configuration Identification:

Define the items that need to be controlled, including code files, documentation, and design specifications.

Create baselines for software components to serve as reference points.

Version Control:

Manage different versions of configuration items.

Track and control changes to the software, ensuring that multiple versions can coexist and be accessed as needed.

Tools: Git, Subversion (SVN), Mercurial.

Change Control:

Evaluate, coordinate, and manage changes to software configuration items.

Use a change control board (CCB) to approve or reject changes.

Ensure that changes are documented and tracked.

Configuration Status Accounting:

Record and report information about the status of configuration items, including the implementation status of change requests.

Provide visibility into the state of the software and its configuration.

Configuration Audits and Reviews:

Conduct audits to verify that configurations conform to specified requirements and standards.

Ensure that documentation is complete and accurate.

Verify that the correct version of a configuration item is being used.

Build Management:

Automate the process of compiling and linking software components to create executable products.

Manage build dependencies and automate build processes.

Tools: Jenkins, Travis CI, Bamboo.

Release Management:

Plan, schedule, and control the movement of releases to test and live environments.

Ensure that the release process is consistent and repeatable.

Tools: Release automation tools, continuous integration/continuous deployment (CI/CD) pipelines.

Version Control

Definition

Version control is a component of SCM that involves the management of changes to documents,

programs, and other information stored as computer files. It allows multiple people to collaborate on projects, track changes, and revert to previous versions when necessary.

Types of Version Control Systems

Local Version Control Systems:

Simple databases that keep all changes to files on a local disk.

Limited collaboration and lack of distributed functionality.

Example: RCS (Revision Control System).

Centralized Version Control Systems (CVCS):

Uses a central server to store all versions of files.

Provides access to multiple users and facilitates collaboration.

Examples: CVS (Concurrent Versions System), Subversion (SVN).

Distributed Version Control Systems (DVCS):

Each user has a complete copy of the entire repository, including the history.

Facilitates offline work and enhances collaboration through easy branching and merging.

Examples: Git, Mercurial.

Types of Project Risks

Project Risks

Project-Specific Risks:

Scope Creep: Uncontrolled changes in project scope can lead to delays and budget overruns.

Schedule Risks: Delays in project timelines due to inaccurate estimation, resource shortages, or unexpected issues.

Cost Risks: Budget overruns due to underestimated costs or unforeseen expenses.

Technical Risks:

Technical Challenges: Issues arising from technical complexities, such as integrating new technologies or dealing with technical debt.

Performance Risks: Risks related to the system not meeting performance requirements.

Obsolescence: Technology or components becoming obsolete during the project lifecycle.

Operational Risks:

Resource Risks: Availability of skilled personnel, tools, and infrastructure.

Communication Risks: Ineffective communication among stakeholders can lead to misunderstandings and project delays.

Process Risks: Inadequate or inefficient processes affecting project execution.

External Risks:

Market Risks: Changes in market conditions that can affect project viability.

Regulatory Risks: Changes in laws and regulations that may impact the project.

Environmental Risks: Natural disasters or environmental changes that can disrupt project activities.

Mitigating Project Risks

Risk Identification: Regularly identify and document potential risks.

Risk Assessment: Analyze the likelihood and impact of each identified risk.

Risk Mitigation: Develop strategies to reduce the likelihood or impact of risks.

Risk Monitoring: Continuously monitor risks and the effectiveness of mitigation strategies.

Contingency Planning: Prepare contingency plans to deal with risks that materialize.

By implementing effective Software Configuration Management practices and understanding the various types of project risks, organizations can improve project outcomes, enhance software quality, and ensure that projects are completed on time and within budget.

Q.11 Discuss COCOMO model in detail. Also explain the term Person Month (PM). **(AKTU 2023-24)**

Ans. COCOMO Model:

The **COCOMO (Constructive Cost Model)** is a software cost estimation model that was developed by **Barry Boehm** in 1981. It is designed to help estimate the effort, cost, and time required to develop a software project based on certain project parameters such as size, complexity, and the development environment. COCOMO is widely used for software project planning and provides a systematic approach for estimating the required effort in terms of **person-months (PM)**, **cost**, and **development time**.

COCOMO is a regression-based model that uses historical project data and the experience of developers to predict the time and effort needed for software development.

Person Month (PM):

The term **Person Month (PM)** is used as a unit of measurement in project estimation to quantify the amount of work required. It is a measure of effort, representing the amount of work one person can complete in one month of full-time work.

- **Definition**: A **Person Month (PM)** refers to the work effort expected from one person over the course of a month (typically 160 to 180 hours of work).
- **Usage in COCOMO**: In the COCOMO model, the total effort required for a project is expressed in **Person Months (PM)**. The formula for calculating the effort (PM) involves multiplying the size of the project (in KLOC) by empirically determined constants and factors that account for the complexity and attributes of the project.
    - For example, if a project is estimated to require 100 PM, it means that one person would need 100 months (or 8 years) to complete the project, assuming that the person works full-time on the project for each month. Alternatively, this could also mean 10 people working for 10 months, or 5 people working for 20 months, etc.