SE-WINTER-22 (IT) SOLVED	
Q.NO	QUESTIONS
Q1 A	Describe and discuss on various types of software myths and the true aspects of these myths?
ANS:	Software engineering is surrounded by various myths or misconceptions that sometimes influence how people perceive the field. Here are a few common software myths and the actual truths behind them: **1. "Once It Works, It's Done" Myth:** • **Myth**: Completion of coding means the software is finished. • **Truth**: Software development involves multiple phases beyond coding, including testing, debugging, maintenance, and updates. **2. "Adding More People Speeds Up Development" Myth:** • **Myth**: Increasing the number of developers accelerates project completion. • **Truth**: More people may introduce communication overheads and complexity, often slowing down development due to coordination efforts. **3. "Testing Can Ensure Bug-Free Software" Myth:** • **Myth**: Thorough testing guarantees bug-free software. • **Truth**: Testing can identify and mitigate bugs, but it's impossible to eliminate all bugs due to the complexity of software. **4. "Estimates are Always Accurate" Myth:** • **Myth**: Project timelines and estimates are always precise. • **Truth**: Software development is inherently unpredictable; estimates are often based on assumptions and may vary due to unforeseen challenges. **5. "Quality Comes from More Features" Myth:** • **Myth**: A software's value increases with more features. • **Truth**: Focusing on essential features and usability often leads to better quality than adding numerous functionalities. **6. "Documentation is Not Necessary" Myth:** • **Myth**: Code should be self-explanatory, eliminating the need for extensive documentation. • **Truth**: While self-explanatory code is valuable, comprehensive documentation aids in understanding, maintaining, and scaling software. **7. "Technology Solves All Problems" Myth:** • **Myth**: Latest technology resolves all software challenges. • **Truth**: Technology is a tool; understanding the problem domain and user needs is essential for effective solutions. **8. "Fixing Bugs is Easy" Myth:** • **Myth**: Bug fixing is a simple and quick task. • **Truth**: Some bugs might be straightforward, but others can be complex, requiring substantial time and effort to resolve. **9. "The Customer Knows What They Want" Myth:** • **Myth**: Customers always have a clear idea of their software requirements. • **Truth**: Requirements often evolve; effective communication and collaboration with customers are vital. **10. "A Perfect Methodology Exists" Myth:** • **Myth**: One methodology (Agile, Waterfall, etc.) suits all projects perfectly. • **Truth**: Different methodologies suit different projects; adaptability and tailoring to project needs are crucial. Understanding these myths and their realities helps in navigating the complexities of software engineering, fostering realistic expectations and effective strategies for successful software development.

Q1 B	Explain software Engineering? Explain the software engineering layers with a neat diagram.

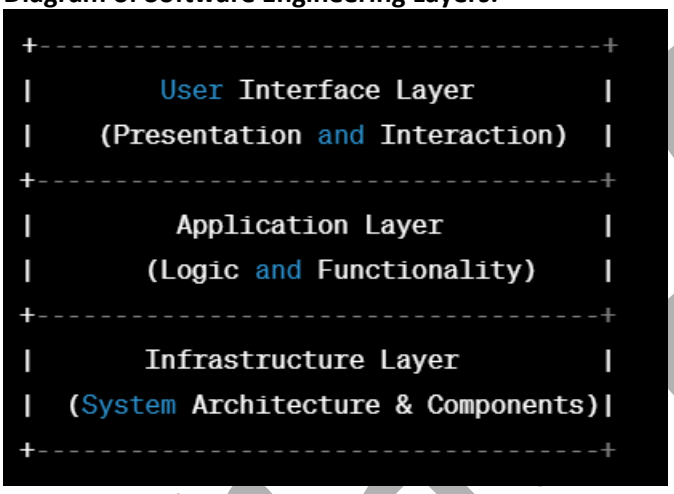
ANS:

Software engineering is a systematic and disciplined approach to designing, developing, and maintaining software systems. It involves applying engineering principles and practices to create reliable, scalable, and maintainable software solutions that meet user needs and business objectives.

Software Engineering Layers:

1. **User Interface Layer**:
   - Concerned with the presentation and interaction aspects of software.
   - Includes GUIs, command-line interfaces, or any user-facing components.
2. **Application Layer**:
   - Encompasses the logic and functionalities of the software.
   - Handles business rules, data processing, algorithms, and core functionalities.
3. **Infrastructure Layer**:
   - Deals with the underlying system architecture and components.
   - Includes databases, servers, operating systems, networking, and hardware.

Diagram of Software Engineering Layers:



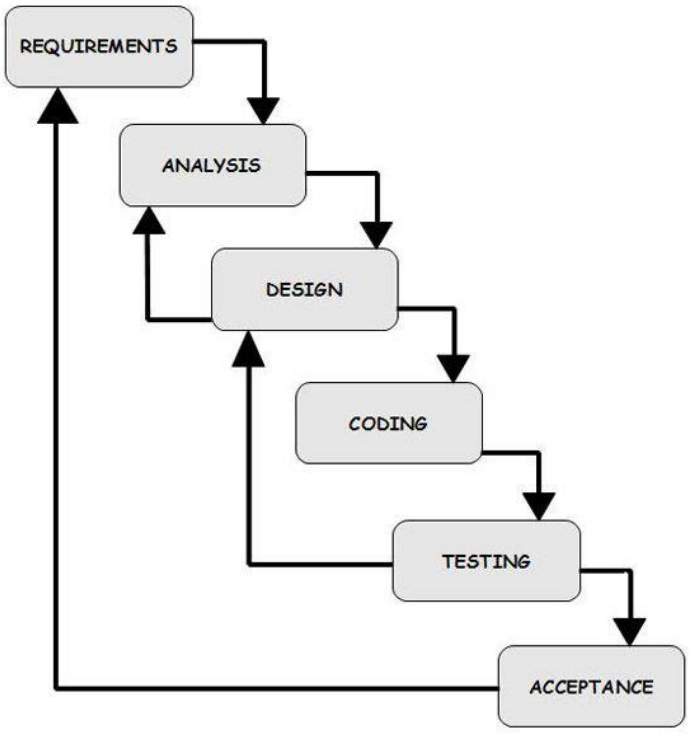
Key Aspects of Each Layer:

1. **User Interface Layer**:
   - Focuses on delivering an intuitive and user-friendly experience.
   - Ensures accessibility, responsiveness, and usability.
2. **Application Layer**:
   - Implements business logic and core functionalities.
   - Manages data, processes, and interactions between different parts of the system.
3. **Infrastructure Layer**:
   - Provides the foundation for software execution.
   - Includes servers, databases, networks, and hardware necessary for the software to operate.

Importance of Layers:

- **Abstraction and Modularity**: Layers facilitate the separation of concerns, enabling modular development and easier maintenance.
- **Scalability and Interoperability**: Clear division allows for scalability and the potential to swap or upgrade components without disrupting the entire system.
- **Focus on Specialization**: Different teams or individuals can specialize in specific layers, enhancing expertise and efficiency.

Software engineering layers help in organizing and conceptualizing software systems, ensuring that each aspect of the software receives appropriate attention and design. This layered approach contributes to building robust, flexible, and manageable software solutions.

Q1 C	Describe with the help of the diagram 'waterfall model'. Give certain reasons for its failure?
ANS:	Waterfall Model is a linear and sequential software development approach that divides the software development lifecycle (SDLC) into distinct phases. Each phase must be completed before moving on to the next, resembling a waterfall cascading through these defined phases. Here's a description along with a diagram and reasons for its failure: **Phases of the Waterfall Model:** 1. **Requirements Gathering**: • Initial phase where project requirements are gathered and documented. 2. **System Design**: • Specification of system architecture and design based on gathered requirements. 3. **Implementation**: • Actual coding or development phase based on the system design. 4. **Testing**: • Verification and validation phase where software is tested for defects or errors. 5. **Deployment**: • Delivery and installation of the software into the operational environment. 6. **Maintenance**: • Post-deployment phase involving software updates, bug fixes, and enhancements. **Diagram of the Waterfall Model:**  **Reasons for Waterfall Model Failure:** 1. **Rigidity and Inflexibility**: • Sequential nature makes it challenging to accommodate changes once a phase is completed. Requirements changes are common, leading to potential rework and delays. 2. **Late Testing**: • Testing occurs after the development phase, which might lead to identification of critical issues at later stages, increasing costs and efforts to fix them. 3. **Limited Customer Interaction**:

	• Minimal customer involvement until the late stages, leading to potential misunderstandings or misinterpretations of requirements. 4. **Assumption of Clear Requirements**: • Assumes that all requirements are known and can be gathered upfront, which might not always be feasible, leading to inadequate understanding or missing requirements. 5. **Risk of Project Failure**: • In case of changes or errors detected late in the process, it becomes difficult and costly to rectify, sometimes leading to project failure. 6. **Not Suitable for Complex Projects**: • Doesn't suit projects with evolving or unclear requirements, intricate designs, or projects requiring continuous feedback and iterations. The Waterfall Model's failure primarily stems from its rigid and linear approach, making it less adaptive to changing requirements and challenging to accommodate customer feedback during early stages. As a result, it's less favoured in today's dynamic and agile environments where flexibility and adaptability are crucial.
Q2 A	List the characteristics of a software. Compare functional requirements with non-functional requirements?
ANS:	**Characteristics of Software:** 1. **Functionality**: • Capability to perform specified tasks and operations. 2. **Reliability**: • Consistency in delivering accurate results under defined conditions. 3. **Usability**: • Ease of use and user-friendliness. 4. **Efficiency**: • Ability to accomplish tasks with minimal resources (time, memory, etc.). 5. **Maintainability**: • Ease of modifying, updating, or enhancing the software. 6. **Portability**: • Ability to function across various environments or platforms. 7. **Scalability**: • Capability to handle increased loads or expand functionalities. 8. **Security**: • Protection against unauthorized access, data breaches, and vulnerabilities. **Functional vs. Non-functional Requirements:** 1. **Functional Requirements**: • Define specific functions or tasks the software should perform. • Examples: User authentication, data processing, report generation. 2. **Non-functional Requirements**: • Define qualities or characteristics the software should possess. • Examples: Performance, reliability, usability, security, scalability. **Comparison:** • **Purpose**: • Functional requirements define what the software should do. • Non-functional requirements define how the software should perform or behave. • **Measurability**: • Functional requirements are generally measurable in terms of features or functionalities. • Non-functional requirements are often qualitative and harder to quantify precisely. • **Impact**: • Functional requirements directly impact user interactions and system capabilities.

	• Non-functional requirements affect overall user experience, system performance, and quality attributes.
	• **Prioritization**:
	  • Functional requirements are usually the primary focus in initial development phases.
	  • Non-functional requirements are critical for ensuring software quality and user satisfaction but often addressed alongside or after functional requirements.
	Both types of requirements are essential for creating a successful software system. Functional requirements drive what the software does, while non-functional requirements govern how well it performs those functions, ensuring the software meets user needs while delivering a high-quality experience.

Q2 B	Discuss the distinct tasks involved in requirement engineering process
ANS:	Requirement Engineering (RE) involves eliciting, analyzing, documenting, validating, and managing software requirements. The distinct tasks in the requirement engineering process include: **1. Elicitation:** • **Objective**: Gather and understand user needs and system requirements. • **Tasks**: Interviews, workshops, surveys, use cases, and observation to collect information from stakeholders. **2. Analysis:** • **Objective**: Analyze gathered requirements for consistency, completeness, and feasibility. • **Tasks**: Use modeling techniques, prioritize requirements, resolve conflicts, and define system boundaries. **3. Specification:** • **Objective**: Document and specify requirements in a clear, concise, and unambiguous manner. • **Tasks**: Create requirement documents, use cases, user stories, diagrams (UML), and other artifacts to capture requirements. **4. Validation:** • **Objective**: Ensure that the specified requirements accurately reflect user needs and expectations. • **Tasks**: Review, verify, and validate requirements with stakeholders, using techniques like prototyping, simulations, or validation meetings. **5. Management and Communication:** • **Objective**: Manage changes to requirements and ensure effective communication among stakeholders. • **Tasks**: Establish a change control process, maintain traceability, and ensure stakeholders are informed about requirement changes. **6. Negotiation and Agreement:** • **Objective**: Resolve conflicts and reach a consensus among stakeholders on conflicting requirements. • **Tasks**: Facilitate discussions, negotiate trade-offs, and reach agreements to resolve conflicting requirements. **7. Documentation:** • **Objective**: Document requirements comprehensively for future reference and understanding. • **Tasks**: Maintain requirement documents, update them as needed, and ensure version control. **8. Quality Assurance:** • **Objective**: Ensure the quality and correctness of the requirements. • **Tasks**: Conduct reviews, perform validation checks, and adhere to industry best practices for requirement engineering. **9. Traceability:** • **Objective**: Establish and maintain traceability between requirements and other artifacts. • **Tasks**: Create traceability matrices to track relationships between requirements, design, and test cases.

	10. Risk Management: • **Objective**: Identify and mitigate risks associated with requirements. • **Tasks**: Analyze potential risks related to requirements, prioritize, and develop strategies to manage or mitigate them. These tasks are iterative and often intertwined throughout the software development lifecycle, ensuring that requirements are well-understood, accurately captured, and effectively managed to meet the stakeholders' needs.
Q2 C	Write the purpose of an SRS? Explain the characteristics of good SRS.
ANS:	The Software Requirements Specification (SRS) is a crucial document that outlines the requirements for a software system to be developed. Its purpose is to serve as a comprehensive reference and communication tool between stakeholders, including developers, testers, project managers, and clients. Here are the key purposes and characteristics of a good SRS: **Purpose of an SRS:** 1. **Capture Requirements**: Document and describe the functional and non-functional requirements of the software system. 2. **Communication**: Serve as a common reference for all stakeholders to understand the system's scope, features, and constraints. 3. **Basis for Development**: Provide a foundation for design, development, testing, and validation activities. 4. **Agreement and Validation**: Ensure that the stakeholders agree on the system requirements before development begins. 5. **Change Control**: Establish a baseline for managing changes to requirements throughout the project lifecycle. 6. **Verification and Validation**: Serve as a basis for verifying that the developed system meets the specified requirements. **Characteristics of a Good SRS:** 1. **Clear and Concise**: Precisely and unambiguously defines the system requirements without unnecessary complexity. 2. **Complete**: Covers all functional and non-functional requirements expected from the system. 3. **Consistent**: Contains requirements that are coherent and not conflicting within the document. 4. **Correct**: Accurately represents the stakeholders' needs and expectations without errors or misunderstandings. 5. **Traceable**: Provides traceability, linking requirements to their origin and to other related documents or artifacts. 6. **Verifiable**: Specifies requirements that are testable and verifiable, ensuring they can be validated against the system's functionality. 7. **Feasible**: Specifies requirements that are feasible within technological, budgetary, and time constraints. 8. **Prioritized**: Clearly defines the importance or priority of each requirement for effective development and testing prioritization. 9. **Maintainable**: Structured and organized for easy maintenance and updates as the project progresses. A well-written SRS sets the foundation for a successful software development project, guiding the team throughout the development lifecycle and serving as a benchmark for measuring the system's adherence to stakeholder expectations.
Q3 A	What are the different types of architectural styles for software? Explain any software architecture in detail?
ANS:	Software architecture defines the structure and behavior of a software system, and various architectural styles provide distinct approaches to organizing and designing systems. Here are some common architectural styles: **1. Layered Architecture:**

	• Organizes the system into logical layers, each handling specific responsibilities. • Example: OSI model, where networking functionalities are divided into layers like application, presentation, and transport. **2. Client-Server Architecture:** • Divides the system into client and server components, where servers provide services to clients. • Examples: Web applications with a browser (client) accessing data from a server via HTTP requests. **3. Microservices Architecture:** • Decomposes the system into small, independent, and loosely-coupled services that communicate via APIs. • Example: Applications built using multiple independently deployable microservices, each focusing on specific functions. **4. Event-Driven Architecture:** • Emphasizes the production, detection, consumption, and reaction to events in a system. • Example: Systems where components communicate through event brokers, reacting to specific events. **5. Component-Based Architecture:** • Structures the system as a collection of reusable, self-contained software components. • Example: Java EE applications using Enterprise JavaBeans (EJBs) as reusable components. **Example of an Architecture: Microservices Architecture** Overview: Microservices architecture breaks down a complex system into smaller, independent services, each focused on a specific business capability and communicating via APIs. Here are its key characteristics: 1. **Decomposition**: • The system is divided into small, independent, and manageable services, each with its own database if needed. 2. **Loose Coupling**: • Services are decoupled, allowing individual services to evolve, deploy, and scale independently. 3. **Service Autonomy**: • Services are owned by individual teams, granting autonomy in development, deployment, and technology stack. 4. **API-First Approach**: • Communication between services is through well-defined APIs, enabling interoperability and independence. 5. **Resilience and Scalability**: • Failure in one service doesn't affect the entire system, and scaling is easier due to the modular nature of services. 6. **Continuous Delivery**: • Facilitates continuous integration and deployment, enabling faster and more frequent releases. **Example Use Case:** Consider an e-commerce platform built using microservices. Each service could handle different functionalities like user management, product catalog, checkout, payment processing, and shipping. Each service operates independently, communicates through APIs, and can be developed, deployed, and scaled separately, providing flexibility and agility in managing the system. Microservices architecture's modularity, scalability, and flexibility make it suitable for complex systems requiring frequent changes, updates, and scalability.
Q3 B	Explain the core activities involved in User Interface design process with necessary block diagram.

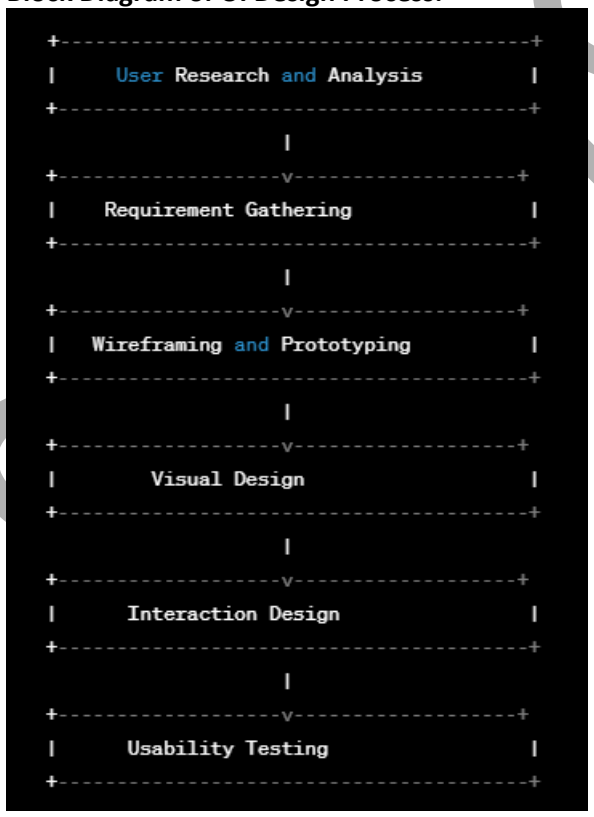
ANS:

The User Interface (UI) design process involves several core activities to create intuitive and user-friendly interfaces. Here are the key activities along with a high-level block diagram:

Core Activities in UI Design Process:

1. **User Research and Analysis**:
   - Gather information about users, their behaviors, needs, and preferences.
   - Analyze user personas, scenarios, and user journeys.
2. **Requirement Gathering**:
   - Understand the project goals, functionalities, and constraints.
   - Define UI requirements based on user and business needs.
3. **Wireframing and Prototyping**:
   - Create low-fidelity wireframes or sketches illustrating layout and structure.
   - Develop interactive prototypes for visualizing and testing the design.
4. **Visual Design**:
   - Apply colors, typography, icons, and images to create an appealing visual interface.
   - Develop style guides to maintain consistency across the UI elements.
5. **Interaction Design**:
   - Design interactive elements such as buttons, menus, forms, and navigation systems.
   - Define user interactions, transitions, and animations.
6. **Usability Testing**:
   - Conduct tests to evaluate the usability and effectiveness of the UI design.
   - Gather feedback from users to identify improvements and enhancements.

Block Diagram of UI Design Process:



Overview of the Block Diagram:

- **User Research and Analysis: Understanding user needs and behaviors.**
- **Requirement Gathering: Defining project and UI requirements.**
- **Wireframing and Prototyping: Creating low-fidelity designs and interactive prototypes.**
- **Visual Design: Adding aesthetics and branding elements.**
- **Interaction Design: Defining user interactions and functionality.**
- **Usability Testing: Evaluating and refining the design based on user feedback.**

	This iterative process ensures that the UI design meets user expectations, aligns with project goals, and provides an engaging and efficient user experience. Each activity contributes to creating a well-crafted interface that addresses user needs while aligning with the project's objectives.
Q3 C	Explain the different types of coupling with examples.
ANS:	In software engineering, coupling refers to the degree of interconnectedness between different modules or components within a system. There are several types of coupling, each indicating the extent to which one module relies on, knows about, or is dependent on another module. Here are some common types of coupling: **1. Content Coupling:** • **Definition**: Content coupling occurs when one module accesses or modifies the internal data or implementation details of another module. • **Example**: Module A directly accesses and modifies variables or data structures within Module B. **2. Common Coupling:** • **Definition**: Common coupling occurs when multiple modules share a global data object or variable. • **Example**: Several modules accessing and updating the same global data structure (e.g., a global variable or file). **3. Control Coupling:** • **Definition**: Control coupling happens when one module controls the behavior of another module by passing control information (e.g., flags or status variables). • **Example**: Module A passes a flag to Module B to influence its behavior or execution flow. **4. Stamp Coupling:** • **Definition**: Stamp coupling occurs when modules exchange a large composite data structure, but only use a portion of that structure. • **Example**: Module A passes an entire data structure to Module B, but Module B only requires a subset of that structure. **5. Data Coupling:** • **Definition**: Data coupling involves modules that are independent but communicate with each other through parameters or shared data. • **Example**: Module A passes necessary data to Module B through function arguments or method parameters. **6. Message Coupling:** • **Definition**: Message coupling occurs when modules communicate by passing messages (often via message queues, interfaces, or APIs). • **Example**: Modules communicate through well-defined interfaces or messaging protocols, sending and receiving messages. **Examples:** • **High Coupling Example**: Module A directly accesses private variables of Module B, tightly coupling their implementations. • **Low Coupling Example**: Module A communicates with Module B through well-defined APIs, with minimal knowledge of its internal workings. Reducing coupling between modules is generally preferred as it enhances maintainability, reusability, and testability of the software system. Loosely coupled systems are more flexible and easier to modify or extend without affecting other parts of the system.
Q4 A	What is Unit testing? Differentiate between black box and white box testing.
ANS:	Unit testing is a software testing method where individual units or components of a software system are tested independently. The objective is to validate that each unit of the software performs as expected and functions correctly in isolation. **Black Box Testing:**

	• **Definition**: Black box testing examines the functionality of a software system without considering its internal code structure or logic. • **Focus**: Focuses on testing inputs and outputs against specifications, without knowledge of the internal workings. • **Approach**: Tests are based on requirements and specifications, simulating user behavior. • **Example**: Testing user interfaces, APIs, or functionalities without knowledge of the underlying code. **White Box Testing:** • **Definition**: White box testing, also known as clear-box or glass-box testing, scrutinizes the internal structure, logic, and code of the software. • **Focus**: Emphasizes testing internal structures, conditions, and code paths for accuracy and completeness. • **Approach**: Tests are designed based on the knowledge of the code structure, execution paths, and internal logic. • **Example**: Code coverage testing, path testing, or examining loops and conditional statements within the code. **Differences:** 1. **Focus**: • Black box testing focuses on the functionality and behavior of the software from an external perspective, based on specifications. • White box testing focuses on the internal structures, logic, and code paths to ensure completeness and accuracy. 2. **Testing Approach**: • Black box testing treats the software as a closed box, testing against specifications and requirements without knowledge of internal workings. • White box testing involves examining the internal structure and code, designing tests based on internal logic and implementation. 3. **Knowledge Requirement**: • Black box testers don't need knowledge of internal code and structures, focusing on user-driven scenarios and specifications. • White box testers require access to the internal code, understanding its structures, logic, and implementation details. 4. **Test Design**: • Black box tests are designed based on inputs, outputs, and user specifications, simulating user interactions. • White box tests are designed based on code paths, internal structures, and implementation details to validate different execution paths. Both black box and white box testing are essential in ensuring comprehensive test coverage. Black box testing verifies the software against user expectations, while white box testing validates internal structures and logic, ensuring code completeness and correctness. Combining both approaches provides more thorough test coverage.
Q4 B	Elaborate on how LOC and FP can be used in project estimation.
ANS:	Lines of Code (LOC) and Function Points (FP) are two common metrics used in software project estimation, each offering distinct approaches to quantify and estimate the size and effort required for software development. **Lines of Code (LOC):** • **Definition**: LOC measures the size of a software project based on the number of lines of code written. • **Estimation Approach**: Estimating effort based on the number of lines of code written or to be written. • **Advantages**:

- Straightforward to measure once the code is written.
- Provides a simple metric for estimating effort and size.

- **Limitations**:
  - Different programming languages may require different LOC for the same functionality.
  - Doesn't consider complexity or functionality; two projects with similar LOC may vary significantly in complexity.

Function Points (FP):

- **Definition**: FP measures the functionality delivered by the software from a user's perspective, irrespective of implementation details.
- **Estimation Approach**: Estimates effort based on the functional requirements and complexity, considering inputs, outputs, inquiries, files, and interfaces.
- **Advantages**:
  - Considers the functionality and complexity delivered to users.
  - Language-independent; doesn't vary with programming languages.
- **Limitations**:
  - Requires detailed understanding and analysis of functional requirements.
  - Complexity in assigning weights to different aspects of the system.

How They Can Be Used in Project Estimation:

1. **LOC in Project Estimation**:
   - Estimating effort based on historical data of lines of code written per unit of time.
   - Calculating estimated effort using a productivity factor (e.g., lines of code per person-month).
2. **FP in Project Estimation**:
   - Analyzing functional requirements to determine Function Point counts.
   - Using a conversion factor (e.g., person-hours per function point) to estimate effort based on FP count.

Comparative Use:

- **LOC**: Suited for simple, well-understood projects with a clear understanding of required code volume.
- **FP**: Ideal for estimating effort based on user-visible functionalities and complexity, irrespective of the implementation language.

Both LOC and FP have their strengths and weaknesses in project estimation. While LOC provides a simple measure based on code volume, FP considers the delivered functionality and complexity, making it more suitable for estimating effort based on user requirements and system complexity. Often, combining multiple estimation techniques provides a more accurate estimation of project effort and size.

Q4 C

Differentiate between (side by side table)
1. verification and validation
2. testing and debugging
3. Alpha testing and Beta testing

ANS:

1. Verification vs. Validation:

Verification	**Validation**
Process of evaluating work products	Process of evaluating the final product
Focuses on ensuring the product meets specifications and standards	Focuses on ensuring the product meets user needs and expectations
Occurs throughout the development phase	Occurs at the end of the development phase
Determines whether the software is being built correctly	Determines whether the right software is being built

Examples: Code reviews, walkthroughs	Examples: User acceptance testing, system testing

2. Testing vs. Debugging:

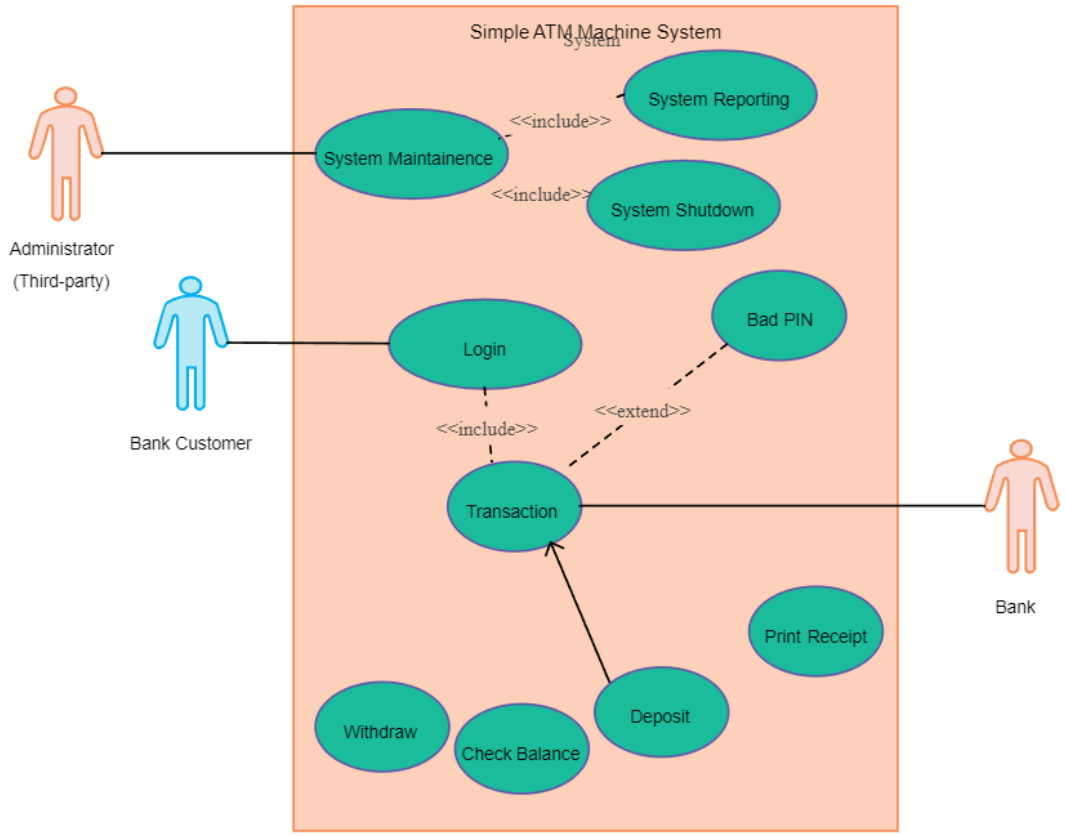
Testing	**Debugging**
Process of evaluating a system or component by executing it under specific conditions	Process of identifying and fixing defects or errors in the system
Aims to identify defects or errors	Aims to eliminate defects or errors
Can be automated or manual	Primarily involves manual intervention
Includes different types (unit, integration, system testing, etc.)	Follows testing to trace and rectify defects
Precedes debugging in the software development lifecycle	Follows testing and precedes final deployment

3. Alpha Testing vs. Beta Testing:

Alpha Testing	**Beta Testing**
Conducted internally by the development team or within the organization	Conducted externally by a select group of end-users or customers
Occurs before beta testing	Occurs after alpha testing and before final release
Aims to identify defects and issues before releasing to beta testers or users	Aims to evaluate product usability, reliability, and functionality
Often not feature-complete	Product is usually feature-complete
Limited users, controlled environment	Larger group of users, real-world scenarios

These distinctions showcase the different stages, objectives, and scopes of these terms within the software development and testing processes.

Q5 A	Draw the use case diagram for ATM system
ANS:	Creating a use case diagram for an ATM system involves identifying the actors (users) interacting with the system and defining the functionalities (use cases) the system provides. Here's a simplified representation:



Actors:

1. **Customer**: Interacts with the ATM system.
2. **Bank System**: Manages authorization, verifies PIN, and handles transactions.

Use Cases (Functionalities):

- **Withdraw**: Allows customers to withdraw money.
- **Deposit**: Allows customers to deposit money.
- **CheckBalance**: Allows customers to check their account balance.
- **Transfer**: Allows customers to transfer funds between accounts.
- **AuthorizeUser**: Part of the bank system, authorizes user access.
- **VerifyPIN**: Part of the bank system, verifies the user's PIN.
- **HandleTransaction**: Part of the bank system, manages the transaction process.

This diagram represents the interaction between the users (Customer) and the ATM System, along with the functionalities provided by the system and interactions with the Bank System to carry out the operations.

Q5 B | Define: Risk. List and explain the phases in risk management.

ANS:

Risk Definition:

Risk in the context of software engineering refers to the potential of an unwanted or adverse event that may have an impact on the successful completion of a project. It involves uncertainties that can affect project objectives, schedules, costs, or quality.

Phases in Risk Management:

1. Risk Identification:
   - **Definition**: Identifying potential risks that could impact the project.
   - **Process**: Brainstorming, documentation review, expert judgment, and historical data analysis.
   - **Objective**: Creating a comprehensive list of risks that the project might face.
2. Risk Analysis:
   - **Definition**: Assessing the identified risks to understand their likelihood and potential impact.

	• **Process**: Qualitative analysis (assessing probability and impact) and quantitative analysis (numerical assessment if possible). • **Objective**: Prioritizing risks based on their severity and likelihood of occurrence. 3. Risk Mitigation Planning: • **Definition**: Developing strategies to minimize or mitigate the impact of identified risks. • **Process**: Creating risk response plans, assigning responsibilities, and defining actions to reduce or eliminate risks. • **Objective**: Planning preventive and corrective actions to address potential risks. 4. Risk Monitoring and Control: • **Definition**: Continuously tracking identified risks, assessing their status, and implementing mitigation plans. • **Process**: Regular review of risks, monitoring changes in risk factors, and ensuring mitigation strategies are effective. • **Objective**: Ensuring that risks are managed throughout the project lifecycle and taking corrective actions when necessary. 5. Risk Communication: • **Definition**: Sharing risk-related information with stakeholders, team members, and decision-makers. • **Process**: Establishing clear communication channels, reporting risk status, and discussing mitigation strategies. • **Objective**: Keeping stakeholders informed about potential risks and the actions taken to manage them. The goal of risk management is to proactively identify, assess, mitigate, and monitor risks throughout the project lifecycle. It's an ongoing process that requires continuous attention to minimize the impact of uncertainties and improve the chances of project success.
Q5 C	Explain i.) Basic principles of software project scheduling. ii.) Quality assurance and Quality Control iii.) Prototyping model
ANS:	**i.) Basic Principles of Software Project Scheduling:** 1. **Define Clear Objectives**: Clearly define project goals, scope, and deliverables. 2. **Task Identification and Breakdown**: Identify all tasks, break them down into smaller, manageable units. 3. **Estimation**: Estimate effort, time, and resources required for each task using techniques like expert judgment, historical data, or software tools. 4. **Sequence Tasks**: Arrange tasks in a logical sequence considering dependencies and constraints. 5. **Allocate Resources**: Assign resources (human, financial, technological) to tasks based on availability and skill set. 6. **Set Milestones and Deadlines**: Establish milestones and deadlines to track progress and ensure timely completion. 7. **Monitor and Adjust**: Continuously monitor progress, adjust schedules based on changes, and manage risks to prevent delays. 8. **Communication and Collaboration**: Maintain open communication among team members, stakeholders, and clients to ensure alignment and manage expectations. **ii.) Quality Assurance (QA) and Quality Control (QC):** **Quality Assurance (QA)**: • **Objective**: Ensures that the processes used to develop and deliver the software are appropriate and effective. • **Activities**: Establishing standards, defining processes, conducting audits, and reviewing methodologies to prevent defects.

- **Focus**: Preventive in nature, aimed at improving processes to avoid issues.

Quality Control (QC):

- **Objective**: Focuses on identifying defects in the actual product.
- **Activities**: Testing, inspecting, and reviewing the software product to identify and rectify defects.
- **Focus**: Corrective in nature, ensuring the product meets defined quality standards.

iii.) Prototyping Model:

Prototyping Model:

- **Definition**: Iterative development approach involving the creation of a prototype to gather user feedback and refine requirements.
- **Process**:
    - **Prototype Development**: Develop a working model of the software based on initial requirements.
    - **User Evaluation**: Present the prototype to users for feedback and requirements validation.
    - **Refinement and Iteration**: Incorporate feedback into the prototype, refine requirements, and iterate the process.
    - **Final Product**: Develop the final product based on refined requirements and user-approved prototype.

Advantages:

- Enhances user involvement and understanding of requirements.
- Early identification of potential issues and changes.
- Improves communication between stakeholders and development teams.

Limitations:

- Can lead to scope creep if not managed properly.
- Might not be suitable for large-scale projects with rigid requirements.

The prototyping model emphasizes user involvement and iterative development to ensure that the final product meets user expectations and requirements through continuous feedback and refinement.