Section A

1. (a) Describe the following characteristics of Software Requirement Specification (SRS) document:

- **(i) Consistent:**
  - An SRS is consistent if there are no contradictions among the requirements.
  - It means no two requirements describe the same thing in different ways or specify conflicting behaviors.
  - For example, if one requirement states a button is green and another states it's blue, the SRS is inconsistent.

- **(ii) Verifiable:**
  - An SRS is verifiable if there exists a cost-effective process to check whether the final software product meets the specified requirements.
  - This implies that requirements are precisely stated, measurable, and testable.
  - Ambiguous terms like "fast" or "user-friendly" make requirements non-verifiable. Instead, quantify them (e.g., "response time less than 2 seconds").
- **(iii) Unambiguous:**

  - An SRS is unambiguous if every requirement has only one possible interpretation.
  - Each statement should be clear, concise, and leave no room for misinterpretation by different stakeholders (developers, testers, users).
  - Using precise language, avoiding jargon where possible, and defining all terms helps in achieving unambiguous requirements.

1. (b) Discuss the advantages of the Incremental process model.

- **Advantages of the Incremental process model:**
  - **Early Delivery of Core Functionality:** Essential system functionalities are delivered early, providing value to the customer sooner.
  - **Reduced Risk:** High-risk functionalities can be developed and tested in earlier increments, allowing for early identification and mitigation of issues.
  - **Customer Feedback:** Regular delivery of increments allows for continuous customer feedback, ensuring the software evolves to meet changing needs.
  - **Manages Complexity:** Breaking down the project into smaller, manageable increments simplifies development and testing.

- **Flexibility:** It is more adaptable to changes in requirements compared to a purely sequential model, as changes can be incorporated in later increments.
- **Faster Deployment:** Each increment is a working system, allowing for quicker deployment and market entry for basic features.

1. (c) What is the difference between conceptual and technical design?

- **Conceptual Design (High-Level Design / Architectural Design):**
    - Focuses on the overall structure and organization of the system.
    - Defines the major components, their responsibilities, and how they interact with each other.
    - Addresses system-wide concerns like security, performance, scalability, and maintainability.
    - It is technology-agnostic in its initial stages, focusing on "what" the system will do and "how" it will be structured at a high level.
    - Aims to bridge the gap between requirements and detailed design.
- **Technical Design (Detailed Design / Low-Level Design):**

- Focuses on the internal logic and implementation details of individual components identified in the conceptual design.
- Specifies data structures, algorithms, interfaces, and module-level design decisions.
- It is technology-specific, taking into account the chosen programming languages, databases, and frameworks.
- Provides the blueprint for coding, detailing "how" each component will be implemented.
- Aims to translate the conceptual design into a concrete, implementable form.

1. (d) What do you mean by design strategy? Discuss the various types of design strategies in brief.

- **Design Strategy:**
  - A design strategy is a systematic approach or plan used to develop the architecture and components of a software system. It guides the design process, helping to make decisions about how the system will be structured, how its parts will interact, and what principles will be followed.
- **Various types of design strategies:**
  - **Function-Oriented Design:**

    - Focuses on the functions or processes that the system performs.
    - The system is decomposed into a set of interacting functions.
    - Data and functions are treated separately.
    - Examples include structured analysis and design (e.g., DFDs and structure charts).

- **Object-Oriented Design (OOD):**
    - Focuses on identifying objects (entities) that encapsulate both data (attributes) and behavior (methods).
    - The system is modeled as a collection of interacting objects.
    - Emphasizes concepts like encapsulation, inheritance, and polymorphism.
    - Aims for reusability, maintainability, and extensibility.

- **Data-Structure Oriented Design:**
    - Focuses on the structure of the data that the system processes.
    - The design of the program structure is derived from the structure of the data.

- Examples include Jackson Structured Programming (JSP) and Warnier/Orr diagrams.

- **Component-Based Design:**
  - Focuses on assembling pre-existing, reusable software components.
  - The system is built by integrating off-the-shelf or custom-built components.
  - Emphasizes modularity, reusability, and faster development.
- **Aspect-Oriented Design (AOD):**
  - Focuses on separating cross-cutting concerns (e.g., logging, security, error handling) from the core business logic.
  - These concerns are encapsulated into "aspects" that can be applied across multiple modules.
  - Aims to improve modularity and maintainability by centralizing concerns that would otherwise be scattered throughout the code.

1. (e) Illustrate the requirement process with the help of a suitable diagram.

- **Illustrating the Requirement Process (in pointers, no diagram):** The requirement process is an iterative cycle that typically involves the following activities:

- **Feasibility Study:**
    - Initial assessment to determine if the proposed system is technically, economically, and operationally feasible.
    - Identifies high-level goals and constraints.
- **Requirements Elicitation (Gathering):**
    - Collecting information from stakeholders about their needs and expectations for the system.
    - Techniques include interviews, surveys, workshops (JAD), brainstorming, prototyping, and use cases.
- **Requirements Analysis and Negotiation:**
    - Analyzing the gathered requirements for consistency, completeness, and ambiguity.
    - Resolving conflicts and prioritizing requirements with stakeholders.
    - Categorizing requirements (functional, non-functional).
- **Requirements Specification:**
    - Documenting the agreed-upon requirements in a formal and structured manner, typically in an SRS document.

    - This document serves as a contract between stakeholders and developers.
- **Requirements Validation:**
    - Reviewing the specified requirements with stakeholders to ensure they accurately reflect their needs and that the system will meet its intended purpose.
    - Techniques include reviews, walkthroughs, and prototyping.
- **Requirements Management:**
    - Managing changes to requirements throughout the software development lifecycle.
    - Maintaining traceability between requirements and other development artifacts (design, code, tests).
    - Controlling versions of requirements.
- **Iteration:** The process is iterative, meaning that activities may be revisited as new information emerges or as the understanding of the system evolves.

1. (f) Consider a software project with the following information domain values :Number of external inputs (I) = 30, Number of external outputs (O) = 60, Number of external inquiries (E) =

23, Number of files (F) = 08, Number of external interfaces (N) = 02. All these values are of average complexity with 4, 5, 4, 10 and 7 respectively. Assume the complexity adjustment value is 1.25. Compute Function Points for the system.

- **Function Points (FP) Computation:**
    a. **Calculate Unadjusted Function Points (UFP):** UFP = (Number of External Inputs * Weight) + (Number of External Outputs * Weight) + (Number of External Inquiries * Weight) + (Number of Files * Weight) + (Number of External Interfaces * Weight)
        - External Inputs (I) = 30, Average Complexity Weight = 4
            - I Contribution = 30 * 4 = 120
        - External Outputs (O) = 60, Average Complexity Weight = 5
            - O Contribution = 60 * 5 = 300
        - External Inquiries (E) = 23, Average Complexity Weight = 4
            - E Contribution = 23 * 4 = 92
        - Files (F) = 08, Average Complexity Weight = 10
            - F Contribution = 8 * 10 = 80

- External Interfaces (N) = 02, Average Complexity Weight = 7
  - N Contribution = 2 * 7 = 14

UFP = 120 + 300 + 92 + 80 + 14 = 606

b. **Apply Complexity Adjustment Value (CAF):** Given Complexity Adjustment Value (CAF) = 1.25.
Function Points (FP) = UFP * CAF

FP = 606 * 1.25 = 757.5

- **Computed Function Points for the system = 757.5**

1. (g) Discuss how the following activities help in achieving high quality for the software:

- **(i) Software Engineering Methods:**
  - **Structured Development:** Methods like structured analysis and design (e.g., DFDs, ERDs, structure charts) promote systematic decomposition, modularity, and clear interfaces, reducing complexity and errors.
  - **Object-Oriented Principles:** Encapsulation, inheritance, and polymorphism lead to more maintainable, reusable, and extensible code, which contributes to higher quality.
  - **Formal Methods:** Using mathematical notation and formal verification techniques can prove the

correctness of critical system components, significantly enhancing reliability and security.

  - **Design Patterns:** Applying proven design patterns provides robust, reusable solutions to common design problems, leading to more reliable and understandable software.
  - **Coding Standards:** Adhering to established coding standards (e.g., naming conventions, code formatting) improves code readability, maintainability, and reduces the likelihood of introducing defects.

- **(ii) Project Management Techniques:**
  - **Planning and Estimation:** Accurate project planning and effort estimation (e.g., using Function Points, COCOMO) ensure adequate resources and time are allocated, preventing rushed development that often compromises quality.
  - **Risk Management:** Proactive identification, assessment, and mitigation of risks (e.g., staff turnover, technology obsolescence) prevent potential issues from negatively impacting product quality or project schedule.
  - **Schedule Tracking and Control:** Monitoring project progress against the schedule and taking corrective

actions ensures that deadlines are met without sacrificing quality.

- **Resource Management:** Effective allocation and utilization of human and technical resources ensure that the right people with the right skills are working on the right tasks, leading to better quality output.
- **Communication and Collaboration:** Establishing clear communication channels and fostering collaboration among team members and stakeholders reduces misunderstandings and ensures everyone is aligned on quality goals.
- **Quality Assurance Activities:** Integrating quality assurance activities (e.g., reviews, audits, testing) throughout the lifecycle, as guided by project management, helps in early defect detection and prevention.

1. (h) Describe the various levels of software testing in brief.

- **Various levels of software testing:**
  - **Unit Testing:**
    - **Purpose:** To test individual components or modules of the software in isolation.
    - **Focus:** Verifying that each unit (e.g., a function, method, or class) performs its intended

functionality correctly according to its design specifications.

  - **Who performs:** Typically performed by developers.
  - **Techniques:** White-box testing (knowledge of internal code structure).

- **Integration Testing:**
  - **Purpose:** To test the interfaces and interactions between integrated units or modules.
  - **Focus:** Ensuring that different modules work together seamlessly and that data flows correctly between them.
  - **Who performs:** Developers or independent testers.
  - **Techniques:** Incremental approaches (top-down, bottom-up, sandwich) using stubs and drivers.

- **System Testing:**
  - **Purpose:** To test the complete, integrated software system against the specified requirements.
  - **Focus:** Verifying that the system meets all functional and non-functional requirements (e.g., performance, security, usability, reliability).

- **Who performs:** Independent testing teams.
- **Techniques:** Black-box testing (no knowledge of internal code).

- **Acceptance Testing:**
  - **Purpose:** To determine if the system meets the customer's business requirements and is ready for deployment.
  - **Focus:** Validating the software against user needs and business processes.
  - **Who performs:** End-users or customer representatives.
  - **Types:** User Acceptance Testing (UAT) and Business Acceptance Testing (BAT).

1. (i) What is coupling? List different types of coupling.

- **Coupling:**
  - Coupling refers to the degree of interdependence between software modules. It measures how strongly one module is connected to, relies on, or interacts with other modules.
  - High coupling indicates strong dependencies, making modules harder to understand, modify, test, and reuse independently.

- Low coupling (loose coupling) is desirable, as it promotes modularity, maintainability, and reusability.

- **Different types of coupling (from strongest to weakest):**
    - **Content Coupling (Pathological Coupling):** One module directly modifies or refers to the internal contents (e.g., data, code) of another module. (Strongest, worst)
    - **Common Coupling (Global Coupling):** Modules share a common global data area (e.g., global variables). Changes to the global data affect all modules using it.
    - **External Coupling:** Modules communicate with an external environment (e.g., hardware, operating system, external files) in a specific format.
    - **Control Coupling:** One module passes control information (e.g., a flag, a function code) to another module, influencing the latter's logic.
    - **Stamp Coupling (Data-Structure Coupling):** Modules share a composite data structure (e.g., an entire record or object) where only a portion of the data is actually needed by the receiving module.
    - **Data Coupling:** Modules communicate by passing only necessary data items as parameters. (Weakest, most desirable)

1. (j) What are umbrella activities? Describe any two umbrella activities.

- **Umbrella Activities:**
  - Umbrella activities are overarching software engineering activities that are applied throughout the entire software development process. They are not tied to a specific phase but rather span across all phases, ensuring the quality, management, and success of the project.
- **Two Umbrella Activities:**
  - **Software Quality Assurance (SQA):**
    - **Description:** SQA is a planned and systematic set of activities ensuring that software processes and products conform to specified requirements, standards, and procedures. It involves monitoring the software engineering process and methods used to ensure quality.
    - **Activities:** Includes defining quality standards, conducting reviews and audits (e.g., formal technical reviews), establishing quality metrics, ensuring adherence to processes, and managing non-conformance. Its goal is to prevent defects from occurring or to detect them early.
  - **Software Configuration Management (SCM):**

        - **Description:** SCM is a discipline that controls the evolution of software systems. It manages changes to software artifacts (requirements, design documents, source code, test cases, etc.) throughout the development lifecycle.
        - **Activities:** Includes version control (tracking changes and maintaining historical versions), change control (managing requests for changes), build management (automating the creation of executable software), and release management (controlling the distribution of software versions). SCM ensures integrity, traceability, and consistency of software artifacts.

Section B

2. (a) Discuss the various process flows with the help of suitable diagrams.

- **Discussing various process flows (in pointers, no diagrams):** Software process flows describe how the activities within a software process are organized and sequenced.
    - **Linear/Sequential Process Flow (e.g., Waterfall Model):**
        - Activities are executed in a strict, sequential manner.

    - Each phase must be completed before the next one begins.
    - There is typically a "go/no-go" decision point at the end of each phase.
    - Suitable for well-understood requirements and stable projects.

- **Iterative Process Flow (e.g., Evolutionary Models like Prototyping, Spiral):**
    - Activities are repeated in cycles or iterations.
    - Each iteration produces a more refined or complete version of the software.
    - Allows for feedback and refinement at various stages.
    - Suitable for projects with evolving requirements or when early user feedback is crucial.
- **Evolutionary Process Flow (e.g., Incremental, Spiral, Agile):**
    - The software evolves over time through multiple iterations.
    - Each iteration adds functionality or refines existing features.
    - Combines elements of linear and iterative flows.

- Emphasizes continuous delivery and adaptation to change.

- **Parallel Process Flow (Concurrent Engineering):**
    - Different activities or sets of activities occur simultaneously or concurrently.
    - For example, design activities for one part of the system might overlap with coding activities for another.
    - Requires careful coordination and communication to manage dependencies.
    - Aims to reduce overall development time.
- **Adaptive Process Flow (e.g., Agile Methodologies like Scrum):**
    - Emphasizes flexibility, rapid response to change, and continuous collaboration.
    - Focuses on short iterations (sprints) with frequent delivery of working software.
    - Requirements and plans can evolve throughout the project.
    - Highly suitable for projects with uncertain or rapidly changing requirements.

2. (b) A Rental Agency needs to develop software for the following requirements: (i) The owner wishing to put his property for rent needs to supply his residence address, telephone number, and the driving license number. (ii) Each owner who registers for this is assigned a unique number (UN) by the computer. (iii) After registration, all the above details are stored in the Owner database. (iv) A similar registration process is undertaken by the potential tenants wishing to rent the property. A tenant can present his UN to the estate agent when he decides to rent the property. (v) The rental agency matches the potential tenant's requirements with the available properties and sends them the details of selected properties. (vi) When a contract between an owner and tenant is completed, the tenant confirms that the contracts have been exchanged and an invoice is sent to the owner and tenant.

Create a Context diagram and level-1 DFD for the Rental Agency.

- **Context Diagram for Rental Agency (in pointers, no diagram):**
    - **System Boundary:** "Rental Agency Software System"
    - **External Entities:**
        - **Owner:** Provides property details, contact info. Receives invoice.

    - **Tenant:** Provides personal details, requirements. Receives property details, invoice.
    - **Estate Agent:** Acts as an intermediary, interacts with both owners and tenants via the system.
  - **Data Flows (to/from System):**
    - From Owner to System: "Owner Registration Details" (Address, Phone, DL Number)
    - From System to Owner: "Owner Unique Number (UN)", "Invoice"
    - From Tenant to System: "Tenant Registration Details", "Tenant Requirements", "Tenant UN for Contract", "Contract Confirmation"
    - From System to Tenant: "Tenant Unique Number (UN)", "Selected Property Details", "Invoice"
    - From Estate Agent to System: (Implicit interaction for matching and contract completion, possibly "Property Matching Request", "Contract Completion Confirmation")
    - From System to Estate Agent: (Implicit "Matching Results", "Contract Status")
- **Level-1 DFD for Rental Agency (in pointers, no diagram):**
  - **Processes (Functions within the System):**

- **1.0 Register Owner:**
  - Input: "Owner Registration Details" (from Owner)
  - Output: "Owner UN" (to Owner)
  - Data Store Interaction: Writes "Owner Details" to "Owner Database"
- **2.0 Register Tenant:**
  - Input: "Tenant Registration Details" (from Tenant)
  - Output: "Tenant UN" (to Tenant)
  - Data Store Interaction: Writes "Tenant Details" to "Tenant Database"
- **3.0 Match Properties:**
  - Input: "Tenant Requirements" (from Tenant)
  - Data Store Interaction: Reads "Property Details" from "Property Database" (assuming properties are registered by owners and stored) and "Tenant Details" from "Tenant Database".
  - Output: "Selected Property Details" (to Tenant)

- **4.0 Manage Contract:**
  - Input: "Tenant UN for Contract" (from Tenant), "Contract Confirmation" (from Tenant - or Estate Agent)
  - Data Store Interaction: Reads "Owner Details" from "Owner Database", "Tenant Details" from "Tenant Database", "Property Details" from "Property Database". Updates "Contract Status" in "Contract Database".
  - Output: "Invoice" (to Owner), "Invoice" (to Tenant)

- **Data Stores:**
  - **Owner Database:** Stores owner's residence address, telephone number, driving license number, and unique number.
  - **Tenant Database:** Stores tenant's registration details and unique number.
  - **Property Database:** (Implicitly needed, stores property details for matching)
  - **Contract Database:** Stores contract details and status.
- **External Entities:** Owner, Tenant, Estate Agent (as defined in Context Diagram).

- **Data Flows:** Connect entities to processes, processes to data stores, and processes to other processes.
  - Owner $\\to$ 1.0: Owner Registration Details
  - 1.0 $\\to$ Owner: Owner UN
  - 1.0 $\\to$ Owner Database: Owner Details
  - Tenant $\\to$ 2.0: Tenant Registration Details
  - 2.0 $\\to$ Tenant: Tenant UN
  - 2.0 $\\to$ Tenant Database: Tenant Details
  - Tenant $\\to$ 3.0: Tenant Requirements
  - 3.0 $\\to$ Tenant: Selected Property Details
  - Property Database $\\leftrightarrow$ 3.0: Property Details (read/write)
  - Tenant Database $\\leftrightarrow$ 3.0: Tenant Details (read)
  - Tenant $\\to$ 4.0: Tenant UN for Contract, Contract Confirmation
  - 4.0 $\\to$ Owner: Invoice
  - 4.0 $\\to$ Tenant: Invoice
  - Owner Database $\\leftrightarrow$ 4.0: Owner Details (read)

        - Tenant Database $\\leftrightarrow$ 4.0: Tenant Details (read)
        - Property Database $\\leftrightarrow$ 4.0: Property Details (read)
        - 4.0 $\\leftrightarrow$ Contract Database: Contract Status (read/write)

3. (a) What is coupling? Discuss any five types of coupling.

- **Coupling:**
    - Coupling refers to the degree of interdependence between software modules. It measures how strongly one module is connected to, relies on, or interacts with other modules.
    - High coupling indicates strong dependencies, making modules harder to understand, modify, test, and reuse independently.
    - Low coupling (loose coupling) is desirable, as it promotes modularity, maintainability, and reusability.
- **Five types of coupling (from strongest to weakest):**
    - **Content Coupling (Pathological Coupling):**
        - **Description:** Occurs when one module directly accesses or modifies the internal data or control flow of another module. This is the highest level of coupling and is highly undesirable.

- **Example:** Module A directly changing a local variable within Module B, or jumping into the middle of a routine in Module B.

- **Common Coupling (Global Coupling):**
  - **Description:** Occurs when two or more modules share access to the same global data structure or global variables. Any change to the global data affects all modules that access it, making it difficult to trace errors and understand dependencies.
  - **Example:** Multiple functions reading from and writing to a shared global configuration object.

- **Control Coupling:**
  - **Description:** Occurs when one module passes a control flag or parameter to another module, which then uses this flag to decide its internal logic or behavior. The calling module dictates the "how" of the called module's operation.
  - **Example:** A function processData(data, typeFlag) where typeFlag determines which specific processing logic processData executes.

- **Stamp Coupling (Data-Structure Coupling):**

        - **Description:** Occurs when modules communicate by passing an entire data structure (e.g., an object, a record) as a parameter, even if the receiving module only needs a small portion of the data within that structure. This exposes unnecessary information and creates dependencies on the internal structure of the data.
        - **Example:** Passing an entire Customer object to a function printCustomerAddress(customer) which only needs the address fields.
    - **Data Coupling:**

        - **Description:** Occurs when modules communicate by passing only elementary data items (e.g., integers, strings, booleans) as parameters. Each parameter is necessary for the receiving module's function. This is the lowest and most desirable form of coupling.
        - **Example:** A function calculateSum(num1, num2) that takes two numbers and returns their sum.

3. (b) What is the purpose of using a structure chart? Prepare a structure chart for the following problem :A system requires a user to login by entering User ID and Password. The system searches for the User ID and the Password for a match. The system allows the user to input the correct password. If the user

enters the correct password, the message "Login Authorised" is output. If the password is incorrect, a warning message "Access Denied" is output.

- **Purpose of using a Structure Chart:**
    - A structure chart is a graphical representation that depicts the hierarchical structure of a software system, showing how modules are organized and how they interact.
    - **Key purposes:**
        - **Modularity:** Illustrates the decomposition of a system into smaller, manageable modules.
        - **Inter-module Relationships:** Shows the calling relationships between modules (who calls whom).
        - **Data Flow:** Indicates the data (parameters) passed between modules, helping to understand coupling.
        - **Control Flow:** Shows control information (flags, switches) passed between modules.
        - **Design Visualization:** Provides a clear, high-level overview of the system's architecture, aiding in design reviews and communication.

- **Maintainability and Testability:** Helps identify areas of high coupling or complexity that might hinder maintainability and testability.

- **Structure Chart for the Login Problem (in pointers, no diagram):**
  - **Main Module:** Login_System
    - **Calls:**
      - Get_User_Credentials (Input: None; Output: User ID, Password)
      - Authenticate_User (Input: User ID, Password; Output: Authentication Status)
      - Display_Login_Result (Input: Authentication Status; Output: None)
  - **Module: Get_User_Credentials**
    - **Purpose:** Prompts the user to enter User ID and Password and returns them.
    - **Inputs:** None
    - **Outputs:** User_ID, Password
  - **Module: Authenticate_User**
    - **Purpose:** Compares the provided User ID and Password against stored credentials.

- **Inputs:** User_ID, Password
- **Outputs:** Authentication_Status (e.g., Boolean: True for authorized, False for denied)
- **Internal Logic:**
    - Searches a data store (e.g., User_Database) for a match.
    - If match found, sets Authentication_Status to True.
    - If no match, sets Authentication_Status to False.

- **Module: Display_Login_Result**
    - **Purpose:** Outputs the appropriate message based on the authentication status.
    - **Inputs:** Authentication_Status
    - **Outputs:** None (displays message)
    - **Internal Logic:**
        - If Authentication_Status is True: Output "Login Authorised".
        - If Authentication_Status is False: Output "Access Denied".

- **Data Flows (represented by arrows with data labels):**
  - User_ID, Password flow from Get_User_Credentials to Login_System.
  - User_ID, Password flow from Login_System to Authenticate_User.
  - Authentication_Status flows from Authenticate_User to Login_System.
  - Authentication_Status flows from Login_System to Display_Login_Result.

4. (a) Draw a Timeline chart for developing an online course registration system for a University.Give three ways of tracking a project schedule.

- **Timeline Chart for Online Course Registration System (in pointers, no diagram):** A timeline chart (or Gantt chart) would visually represent tasks, their durations, dependencies, and progress over time.
  - **Phase 1: Requirements Gathering & Analysis (e.g., Weeks 1-4)**
    - Task: Elicit user requirements (students, faculty, admin).
    - Task: Analyze and document functional/non-functional requirements.

    - Task: Create SRS document.

- **Phase 2: Design (e.g., Weeks 5-8)**
    - Task: Architectural design (system components, database schema).
    - Task: Detailed design (module specifications, UI/UX design).
    - Task: Design database tables.
- **Phase 3: Development (e.g., Weeks 9-20)**
    - Task: Database implementation.
    - Task: Backend module development (e.g., user authentication, course catalog, registration logic).
    - Task: Frontend development (e.g., student portal, admin dashboard).
    - Task: API integration.
- **Phase 4: Testing (e.g., Weeks 21-24)**
    - Task: Unit testing.
    - Task: Integration testing.
    - Task: System testing (functional, performance, security).
    - Task: User Acceptance Testing (UAT).

- **Phase 5: Deployment & Maintenance (e.g., Weeks 25 onwards)**
  - Task: System deployment.
  - Task: User training.
  - Task: Post-deployment support and bug fixing.
- **Dependencies:** Requirements must be largely complete before design. Design must be largely complete before development. Development must be complete before testing.
- **Milestones:** SRS Approved, Design Approved, Core Modules Coded, System Testing Complete, UAT Sign-off, System Live.

- **Three ways of tracking a project schedule:**
  - **Gantt Charts (or Timeline Charts):**
    - Visually represent tasks, their durations, start/end dates, and dependencies.
    - Progress can be tracked by shading or coloring bars to show completion percentage.
    - Allows for easy identification of critical path tasks and schedule slippage.
  - **Earned Value Management (EVM):**

- Integrates scope, schedule, and cost to provide an objective measure of project performance.
- Compares the planned value of work (Planned Value - PV), the actual cost of work performed (Actual Cost - AC), and the earned value of work performed (Earned Value - EV).
- Provides metrics like Schedule Variance (SV = EV - PV) and Schedule Performance Index (SPI = EV / PV) to quantify schedule adherence.

- **Milestone Tracking:**
  - Identifies key checkpoints or significant events in the project lifecycle (milestones).
  - The project schedule is tracked by monitoring whether these milestones are achieved on time.
  - Provides a high-level view of progress and helps in identifying major delays.

4. (b) Differentiate between the following:

- **(i) Stub and driver:**
  - **Stub:**
    - A dummy program or module that simulates the functionality of a lower-level module that is not yet developed or integrated.

        - Used in top-down integration testing to allow testing of higher-level modules without waiting for the completion of all dependent lower-level modules.
        - It typically has the same interface as the actual module but provides simplified or predefined responses.
    - **Driver:**
        - A dummy program or module that simulates the functionality of a higher-level module that calls the module being tested.
        - Used in bottom-up integration testing to provide the necessary environment (inputs, calls) for testing a lower-level module in isolation.
        - It calls the module under test and passes test data to it.
- **(ii) Verification and validation:**
    - **Verification (Are we building the product right?):**
        - The process of evaluating whether a product, service, or system complies with a regulation, requirement, specification, or imposed condition.

    - It focuses on checking if the software meets its specifications and if the development process is being followed correctly.
    - Typically involves reviews, inspections, walkthroughs, and static analysis.
    - Performed throughout the development lifecycle (e.g., design verification, code verification).
  - **Validation (Are we building the right product?):**
    - The process of evaluating whether a product, service, or system satisfies the needs of the customer and other identified stakeholders.
    - It focuses on ensuring that the software meets the actual business needs and user expectations.
    - Typically involves dynamic testing, user acceptance testing (UAT), and system testing.
    - Performed towards the end of the development lifecycle.
- **(iii) Error and defect:**
  - **Error (Human Mistake):**
    - A human action that produces an incorrect result. It's a mistake made by a person (e.g., a developer, a requirements analyst) during any phase of the software development lifecycle.

    - Errors can lead to defects.
    - Example: A developer misunderstanding a requirement and writing incorrect code.
  - **Defect (Fault/Bug):**
    - An imperfection or a flaw in a software component or system that can cause the system to fail to perform its required function. It is the manifestation of an error in the code or documentation.
    - A defect is what is found during testing.
    - Example: A line of code that produces an incorrect calculation, or a missing feature as per the specification.

- **(iv) White-box and black-box testing:**
  - **White-box Testing (Structural Testing/Glass-box Testing):**
    - **Approach:** Tests the internal structure, design, and implementation of the software system. The tester has full knowledge of the source code, internal logic, and architecture.
    - **Focus:** Verifying the internal workings of a module, including code paths, loops, conditions, and data flow.

- **Techniques:** Path testing, statement coverage, branch coverage, condition coverage.
- **When used:** Primarily at unit and integration testing levels.

- **Black-box Testing (Functional Testing/Behavioral Testing):**
  - **Approach:** Tests the functionality of the software system without any knowledge of its internal structure or code. The tester interacts with the software through its external interfaces, treating it as a "black box."
  - **Focus:** Verifying that the software meets its specified requirements and behaves as expected from a user's perspective.
  - **Techniques:** Equivalence partitioning, boundary value analysis, decision table testing, state transition testing, use case testing.
  - **When used:** Primarily at system and acceptance testing levels.

5. (a) Draw the risk table with at least three entries and discuss its components in brief. Make the necessary assumptions.

- **Risk Table (in pointers, no diagram):** A risk table is a tool used in risk management to document identified risks and their attributes.

  Assumptions:

  - This is for a typical software development project.
  - The probability is rated on a scale (e.g., Low, Medium, High or 1-5).
  - The impact is rated on a scale (e.g., Low, Medium, High or 1-5).
  - Risk exposure is calculated as Probability x Impact.

  Risk Table Entries (Example):

Risk ID	Risk Description	Probability	Impact	Risk Exposure (P x I)	Mitigation Strategy	Contingency Plan
R01	Staff Turnover	High (0.7)	High (4)	2.8	Cross-training, retention programs, knowledge transfer.	Hire new staff, reassign tasks, extend schedule.

Risk ID	Risk Description	Probability	Impact	Risk Exposure (P x I)	Mitigation Strategy	Contingency Plan
R02	Requirements Volatility	Medium (0.5)	Medium (3)	1.5	Incremental development, prototyping, frequent stakeholder meetings.	Re-plan affected iterations, negotiate scope changes.
R03	Technology Obsolescence	Low (0.2)	High (4)	0.8	Research new tech, build POCs, modular architecture.	Re-architect affected components, use alternative tech.

- **Components of a Risk Table:**

- **Risk ID:** A unique identifier for each risk.
- **Risk Description:** A clear and concise statement of the risk, often in an "If <cause>, then <event>, which leads to <consequence>" format.
- **Probability (P):** The likelihood of the risk event occurring. This can be qualitative (e.g., Low, Medium, High) or quantitative (e.g., a percentage or a value from 0.1 to 1.0).
- **Impact (I):** The severity of the consequences if the risk event occurs. This can be qualitative (e.g., Low, Medium, High) or quantitative (e.g., a cost in dollars, a delay in days, or a value from 1 to 5). Impact can be on schedule, budget, quality, or resources.
- **Risk Exposure (or Risk Level):** A calculated value (often Probability x Impact) that indicates the overall significance of the risk. It helps in prioritizing risks.
- **Mitigation Strategy:** Actions taken to reduce the probability of the risk occurring or to reduce its impact if it does occur. These are proactive measures.
- **Contingency Plan:** Actions to be taken if the risk event actually occurs, to minimize its negative effects. These are reactive measures, a "plan B."

5. (b) Find the cyclomatic complexity of the graph given below using three different methods.

- **Cyclomatic Complexity Calculation (in pointers, no diagram):** The problem states "a graph with 5 nodes and 6 edges." Let's assume a generic graph structure that allows for calculation (e.g., a connected graph with a single entry and single exit point if it represents a flow graph).
    - **Method 1: Using the formula V = E - N + 2**
        - E = Number of Edges = 6
        - N = Number of Nodes = 5
        - V = Cyclomatic Complexity
        - V = 6 - 5 + 2 = 3
        - **Cyclomatic Complexity = 3**
    - **Method 2: Using the formula V = P + 1 (where P is the number of predicate nodes)**
        - Predicate nodes are nodes that have more than one outgoing edge (decision points).
        - Without the actual diagram, I must assume the number of predicate nodes. For a cyclomatic complexity of 3 with 5 nodes and 6 edges, a common structure would be 2 predicate nodes.
        - Let's assume there are 2 predicate nodes in the graph.
        - P = 2

        - V = 2 + 1 = 3
        - **Cyclomatic Complexity = 3**
    - **Method 3: Using the number of regions**
        - A region is an area bounded by edges and nodes in the graph. The "outside" of the graph is also counted as a region.
        - For a planar graph (which most flow graphs are), the cyclomatic complexity is equal to the number of regions (R) created by the graph when drawn on a plane.
        - With 5 nodes and 6 edges, a typical graph that yields a complexity of 3 would have 2 inner regions and 1 outer region.
        - R = 2 (inner regions) + 1 (outer region) = 3
        - **Cyclomatic Complexity = 3**

6. (a) How can Risk Mitigation, Monitoring, and Management (RMMM) be applied to "staff turnover" risk? Discuss in brief.

- **Applying Risk Mitigation, Monitoring, and Management (RMMM) to "Staff Turnover" Risk:**
    - **Risk Mitigation:** (Proactive actions to reduce probability or impact)

- **Competitive Compensation & Benefits:** Ensure salaries and benefits are competitive with industry standards to reduce financial motivation for leaving.
- **Positive Work Environment:** Foster a supportive, collaborative, and respectful work culture.
- **Career Development & Training:** Provide opportunities for skill enhancement, professional growth, and clear career paths.
- **Work-Life Balance:** Promote policies that support employee well-being, such as flexible hours or remote work options.
- **Recognition & Rewards:** Implement programs to acknowledge and reward employee contributions.
- **Effective Leadership:** Train managers to be good leaders, providing clear direction, constructive feedback, and support.
- **Knowledge Transfer & Cross-training:** Proactively cross-train team members on different modules/tasks and ensure knowledge is documented to reduce impact if someone leaves.

- **Risk Monitoring:** (Tracking the risk and mitigation effectiveness)
    - **Employee Satisfaction Surveys:** Regularly conduct surveys to gauge morale, identify pain points, and assess job satisfaction.
    - **Exit Interviews:** Conduct structured interviews with departing employees to understand reasons for leaving and identify recurring issues.
    - **Turnover Rate Tracking:** Monitor the project's and organization's staff turnover rates against industry benchmarks.
    - **Performance Reviews:** Use regular performance reviews as an opportunity to discuss career aspirations and address concerns.
    - **Informal Check-ins:** Encourage managers to have regular informal conversations with team members to build rapport and identify potential dissatisfaction early.
- **Risk Management (Contingency Planning):** (Reactive actions if the risk materializes)
    - **Succession Planning:** Identify potential replacements for critical roles and prepare them for transition.

- **Talent Pipeline:** Maintain a pool of potential candidates (e.g., through recruitment agencies, university partnerships) for rapid hiring.
- **Task Reassignment:** If a key person leaves, quickly reassign their critical tasks to other team members or new hires.
- **Schedule Adjustment:** If staff turnover significantly impacts progress, adjust project schedules and communicate changes to stakeholders.
- **Temporary Staffing:** Consider bringing in contractors or temporary staff to fill immediate gaps.
- **Knowledge Base Utilization:** Rely on well-documented knowledge bases and cross-training efforts to minimize disruption.

7. (a) Discuss any seven elements of Software Quality Assurance (SQA).

- **Seven Elements of Software Quality Assurance (SQA):**
  - **1. SQA Plan:**
    - A document that outlines the quality objectives, activities, responsibilities, and resources for a

software project. It defines how quality will be ensured throughout the lifecycle.

- **2. Reviews and Audits:**
    - Systematic examinations of software products and processes by peers or independent teams to identify defects, deviations from standards, and areas for improvement. Includes formal technical reviews, walkthroughs, and inspections.
- **3. Software Testing:**
    - A planned process of executing a program with the intent of finding errors. SQA ensures that appropriate testing strategies (unit, integration, system, acceptance) are applied at each stage and that test coverage is adequate.
- **4. Error/Defect Tracking and Analysis:**
    - Establishing a system to log, categorize, prioritize, track, and resolve identified defects. Analyzing defect data helps in identifying root causes and improving future processes.
- **5. Change Management (Configuration Management):**
    - Controlling changes to software artifacts (requirements, design, code, tests) to ensure

consistency and traceability. SQA ensures that changes are properly reviewed, approved, and implemented.

- **6. Standards and Procedures:**
  - Defining and enforcing software development standards (e.g., coding standards, documentation standards) and procedures (e.g., coding guidelines, review checklists) to ensure consistency and quality across the project.
- **7. Measurement and Metrics:**
  - Collecting and analyzing data related to software quality (e.g., defect density, test coverage, cyclomatic complexity) to quantitatively assess product and process quality, identify trends, and support decision-making.
- **8. Training:**
  - Providing necessary training to development and QA teams on processes, tools, standards, and specific technologies to enhance their skills and improve the quality of their work.
- **9. Risk Management:**

    - Integrating risk identification, assessment, and mitigation into the SQA process to proactively address potential threats to software quality.

7. (b) Briefly explain the following:

- **(i) Scrum:**
    - **Description:** Scrum is an agile framework for managing complex projects, particularly software development. It is iterative and incremental, focusing on delivering working software frequently.
    - **Key Concepts:**
        - **Sprints:** Short, time-boxed iterations (typically 1-4 weeks) during which a "Done," usable, and potentially releasable product increment is created.
        - **Product Backlog:** An ordered list of all known product requirements and features, prioritized by the Product Owner.
        - **Sprint Backlog:** A subset of the Product Backlog selected for a specific sprint, broken down into tasks.
        - **Daily Scrum:** A short daily meeting (15 minutes) where the Development Team synchronizes activities and plans for the next 24 hours.

        - **Roles:** Product Owner (defines "what"), Scrum Master (facilitates the process), Development Team (builds "how").
        - **Events:** Sprint Planning, Daily Scrum, Sprint Review, Sprint Retrospective.

- **(ii) Use cases:**
    - **Description:** A use case is a technique used in software engineering to describe the functional requirements of a system from the perspective of an external user (actor) interacting with the system. It captures a sequence of actions that provide a measurable value to an actor.
    - **Key Components:**
        - **Actor:** A person, system, or organization that interacts with the system.
        - **Goal:** The objective the actor wants to achieve.
        - **Preconditions:** Conditions that must be true before the use case can start.
        - **Postconditions:** Conditions that are true after the use case successfully completes.
        - **Main Flow (Scenario):** The typical sequence of steps taken by the actor and the system to achieve the goal.

  - **Alternative Flows:** Variations or exceptions to the main flow (e.g., error handling, optional steps).

- **Purpose:** Use cases help in understanding, documenting, and communicating system requirements in a user-centric way, facilitating communication between stakeholders and developers.

7. (b) Consider a simple program to classify a triangle. Its inputs is a triple of positive integers (say x, y, z) and the date type for input parameters ensures that these will be integers greater than 0 and less than or equal to 100.The program output may be one of the following words :[Scalene; Isosceles; Equilateral; Not a Triangle].

Design the Boundary Value test cases.

- **Boundary Value Test Cases for Triangle Classification:**

  The input parameters are x, y, z where $0 \< x, y, z \\le 100$. Boundary values for each side are:

  - Minimum Valid: 1
  - Maximum Valid: 100
  - Values just inside the boundary: 2, 99
  - Values just outside the boundary (for invalid cases): 0, 101 (though problem states "integers greater than 0

and less than or equal to 100" so these might be caught by input validation before classification logic).

We also need to consider the triangle inequality theorem: the sum of the lengths of any two sides of a triangle must be greater than the length of the third side (a + b > c, a + c > b, b + c > a).

Test Cases:

c. **Equilateral Triangle (All sides equal):**

- (1, 1, 1) - Minimum boundary
- (50, 50, 50) - Mid-range
- (100, 100, 100) - Maximum boundary

d. **Isosceles Triangle (Two sides equal):**

- (1, 1, 2) - Not a Triangle (boundary for inequality)
- (2, 2, 1) - Valid Isosceles (min boundary for two sides, one smaller)
- (2, 1, 2) - Valid Isosceles
- (1, 2, 2) - Valid Isosceles
- (99, 99, 1) - Valid Isosceles (large equal sides, small third)

- (100, 100, 99) - Valid Isosceles (max equal sides, one smaller)
- (100, 100, 100) - Already covered by Equilateral, but important to note.
- (1, 99, 99) - Valid Isosceles (small first side, large equal)
- (99, 1, 99) - Valid Isosceles
- (99, 99, 100) - Valid Isosceles (large equal sides, one larger)
- (100, 99, 99) - Valid Isosceles
- (99, 100, 99) - Valid Isosceles

e. **Scalene Triangle (All sides different):**

- (2, 3, 4) - Valid Scalene (min boundary values)
- (1, 2, 3) - Not a Triangle (boundary for inequality)
- (98, 99, 100) - Valid Scalene (max boundary values)
- (10, 20, 25) - Mid-range

f. **Not a Triangle (Violating Triangle Inequality):**

- (1, 2, 3) - Sum of two sides equals third (a+b=c)

- (1, 2, 4) - Sum of two sides less than third (a+b<c)
- (10, 20, 30) - Sum of two sides equals third
- (10, 20, 31) - Sum of two sides less than third
- (100, 1, 1) - Extreme case (very long side, two very short)
- (1, 100, 1)
- (1, 1, 100)

g. **Invalid Input (based on data type constraints, though problem states input parameters ensure validity):**

- (0, 50, 50) - Value less than or equal to 0 (if not caught by input validation)
- (101, 50, 50) - Value greater than 100 (if not caught by input validation)
- (-1, 50, 50) - Negative value (if not caught by input validation)

This set covers boundaries for individual side lengths and for the triangle inequality condition, aiming to test the limits of the input domain and the logic for triangle classification.