UNIT 1

1. What is Software Engineering? What is the role of a software engineer? Compare Hardware and Software product characteristics.

- **Software Engineering** is the process of designing, developing, testing, and maintaining software systems in a systematic and efficient way.
- **Role of Software Engineer**: Involves writing and testing code, designing system architectures, troubleshooting issues, and ensuring the software meets user requirements.
- **Hardware vs. Software**:
    - **Hardware**: Physical, tangible, often fixed in design, requires physical resources for production.
    - **Software**: Intangible, can be easily modified and updated, requires no physical resources for replication, and is scalable.

2. Explain Software Engineering as a Layered Technology.

Software engineering is viewed as a multi-layered process that includes:

- **Process Layer**: Methods and procedures used in software development.
- **Methods Layer**: Techniques used for designing and developing software (e.g., algorithms, architecture design).
- **Tools Layer**: Software tools that support development (e.g., IDEs, version control systems).
- **Quality Layer**: Ensures the software meets required standards.

3. Explain software myths for customers, practitioners, and project managers. What is the reality in each case?

- **Customer Myths**:
  - Myth: "The software can do anything once developed."
  - Reality: Software requires ongoing maintenance and adjustments after delivery.
- **Practitioner Myths**:
  - Myth: "The best developers can code without any specifications."
  - Reality: Clear requirements and communication are essential for successful projects.
- **Project Manager Myths**:
  - Myth: "Software projects are predictable once planning is done."
  - Reality: Software development is inherently uncertain and requires flexibility.

4. Describe generic view of software engineering or List and explain very briefly various activities of software engineering process framework or Explain Umbrella activities and its role in SDLC.

- **Generic view**: The software engineering process includes:
  - **Requirements Gathering**
  - **Design**
  - **Implementation**
  - **Testing**
  - **Deployment**

    - **Maintenance**

- **Umbrella Activities**: These are activities that overlap and support various stages like configuration management, project management, quality assurance, etc.

5. Distinguish between a program and a software product.

- **Program**: A set of instructions written to perform a specific task, typically smaller in scale.
- **Software Product**: A complete solution that meets user needs, often complex and contains features such as documentation, maintenance, and support.

6. What do you mean by software model? Explain each model in detail.

A **software model** is an abstraction or representation of the software development process. Common models:

- **Waterfall Model**: Linear, sequential approach.
- **Incremental Model**: Breaks the software into smaller, more manageable pieces.
- **Spiral Model**: Focuses on iterative development with risk analysis.
- **Agile Model**: Emphasizes flexibility and customer collaboration.

7. Differentiation between incremental process model and waterfall model.

- **Incremental Model**: Develops the software in parts, adding features iteratively.
- **Waterfall Model**: Follows a linear, sequential approach, with distinct phases that don’t overlap.

8. Differentiation between prototype model and waterfall model.

- **Prototype Model**: Involves creating a basic version of the software early on and refining it based on user feedback.
- **Waterfall Model**: Linear and rigid with no scope for changes once a phase is completed.

9. Differentiation between prototype model and RAD model.

- **Prototype Model**: Focuses on quickly building a prototype to gather user feedback.
- **RAD Model**: Involves rapid prototyping and focuses on quick development and iteration, but typically with more emphasis on end-user customization.

10. Differentiation between prototype model and spiral model.

- **Prototype Model**: Emphasizes quick creation and feedback from early-stage prototypes.
- **Spiral Model**: Combines iterative development with risk management, focusing on incremental builds.

11. Explain Spiral model with suitable example. Also explain how it differs from Software Prototyping model.

- The **Spiral Model** involves cyclic development with a focus on risk analysis and iteration, and it typically involves four phases: planning, risk analysis, engineering, and evaluation.
- Difference from Prototyping: The Spiral Model emphasizes risk management, while the Prototype Model focuses on user feedback and rapid development.

12. Explain the process model which is used in situations where the requirements are well-defined.

- The **Waterfall Model** is typically used when the requirements are well-defined and unlikely to change during the development process.

13. Explain List different agile process models. Explain any one with suitable example.

- Agile models include:
  - **Scrum**
  - **Extreme Programming (XP)**
  - **Lean Software Development**
  - **Feature-Driven Development (FDD)**
- **Example (Scrum)**: Scrum focuses on iterative development in fixed-length sprints with roles like Product Owner, Scrum Master, and Development Team.

14. Myths of planned development.

- **Myth**: "Planning solves all development problems."
- **Reality**: Software development is dynamic and planning needs to be adaptive to changes.

15. Explain Extreme Programming process.

- **Extreme Programming (XP)** emphasizes customer satisfaction, frequent releases, pair programming, and continuous feedback.

16. What is Scrum? Explain in detail.

- **Scrum** is an agile framework that divides work into sprints (2-4 weeks), with a focus on delivering a working product increment after each sprint. It includes daily stand-up meetings, sprint reviews, and retrospectives.

17. What are agile practices? Explain each in detail.

Key agile practices include:

- **Continuous Integration**: Merging code frequently to avoid integration issues.
- **Test-Driven Development (TDD)**: Writing tests before code to ensure quality.
- **Pair Programming**: Two developers work together at one computer to write code.
- **User Stories**: Brief descriptions of features written from the user's perspective.

18. Discuss agile modeling significance. What are different modeling principles that make agile modeling unique?

- **Agile Modeling** emphasizes simplicity, flexibility, and collaboration. Principles include:
    - **Model with a purpose**: Models are only created if they add value.
    - **Embrace change**: Models should be flexible and updated regularly.

19. Define agility. What are the things that agile software should have?

- **Agility** refers to the ability to adapt quickly to changes and continuously improve. Agile software should be flexible, maintainable, collaborative, and able to deliver value quickly.

20. Explain Adaptive Software Development model in detail.

- The **Adaptive Software Development (ASD)** model focuses on flexibility and iteration. It includes three primary phases:

Speculation (planning), Collaboration (development), and Learning (review and refinement).

UNIT 2

1. What do you mean by risk? What is software risk? Explain all types of software risk.

- **Risk** refers to the possibility of an event occurring that could negatively impact the outcome of a project. It is associated with uncertainty and can affect the project's cost, time, or quality.
- **Software Risk**: In the context of software engineering, it refers to the potential for problems or issues that can affect the software development process, its success, or its performance. These risks can arise during development, deployment, or operation.

Types of Software Risks:

1. **Project Risks**: Risks that affect the overall success of the project (e.g., deadlines, cost overruns).
2. **Technical Risks**: Risks related to the technology, such as feasibility, performance, or compatibility issues.
3. **People Risks**: Risks related to the people involved in the project, including skill shortages, miscommunication, or turnover.
4. **External Risks**: Risks that come from outside the project, like changes in market conditions, legal requirements, or technology shifts.
5. **Operational Risks**: Risks related to the operational environment or usage of the software, like security vulnerabilities or system failures.

2. Write short note on: Risk Management or RMMM

- **Risk Management (RMMM)** is the process of identifying, analyzing, and mitigating risks throughout a software project. It

involves planning for potential issues and ensuring that strategies are in place to handle them effectively.

- The **Risk Management Plan (RMMM)** includes:
    - **Risk Identification**: Identifying potential risks that could impact the project.
    - **Risk Assessment**: Analyzing the likelihood and impact of each risk.
    - **Risk Mitigation**: Developing strategies to reduce or avoid risks.
    - **Risk Monitoring**: Continuously tracking and adjusting the plan as new risks emerge.

3. Explain Software Project Management and W5HH principles.

- **Software Project Management**: It involves the planning, coordination, and execution of a software project. The goal is to ensure that the project is completed on time, within budget, and meets the required quality standards. This includes tasks like defining scope, managing resources, tracking progress, and handling risks.

W5HH Principles: These principles guide project management decisions and communication:

1. **What**: Define the objectives and deliverables of the project.
2. **Why**: Understand the reason or purpose behind the project.
3. **When**: Establish timelines and milestones.
4. **Where**: Determine where the project will be carried out (e.g., teams, tools, infrastructure).
5. **How**: Plan the methodologies and processes to be followed.

6. **How Much**: Define the budget and resources required for the project.

4. What is software measurement? Explain Software metrics used for software cost estimation. Or Explain Software metrics in detail.

- **Software Measurement** refers to the process of quantifying software attributes, such as size, complexity, performance, and quality, to manage and improve the software development process.
- **Software Metrics** are key for tracking project progress, quality, and performance. They can help in estimating software costs, evaluating productivity, and ensuring that the development meets required standards.

Software Metrics for Cost Estimation:

1. **Function Points**: Measures the size of software based on its functionality (inputs, outputs, inquiries, files, etc.). This helps estimate the effort required for development.
2. **Lines of Code (LOC)**: Measures the size of the software in terms of the number of lines of code written. It helps estimate the effort, cost, and time to develop.
3. **Cyclomatic Complexity**: Measures the complexity of a program based on the number of decision points in the code. This can indicate the difficulty and risk of maintaining the software.
4. **Cost per Function Point**: Helps estimate the cost of the project by calculating the cost for each function point in the system.
5. **Development Time**: Estimates how much time it will take to develop the software based on metrics like function points, LOC, or past project data.

5. Explain Software Project Planning.

- **Software Project Planning** is the process of defining the goals, tasks, resources, schedule, and deliverables for a software project. It provides a roadmap for managing and executing the project and helps in tracking progress, managing risks, and ensuring that the project meets its objectives.

Key elements of **Software Project Planning** include:

1. **Defining Scope**: Identifying the project's objectives and deliverables.
2. **Work Breakdown Structure (WBS)**: Breaking down the project into smaller, manageable tasks.
3. **Resource Allocation**: Assigning resources such as personnel, tools, and equipment to tasks.
4. **Scheduling**: Developing a timeline and setting milestones.
5. **Risk Management**: Identifying potential risks and defining strategies to mitigate them.
6. **Cost Estimation**: Estimating the budget required to complete the project.

6. Explain Project Scheduling Process and Gantt Chart in detail.

- **Project Scheduling** is the process of determining when tasks need to be completed, assigning resources, and setting milestones to ensure the project stays on track. The goal is to optimize the use of resources and time while meeting deadlines.

Gantt Chart: A Gantt Chart is a type of bar chart used to represent a project schedule. It visually shows the start and end dates of each task, the dependencies between tasks, and the overall timeline of the project. It helps project managers track progress, manage resources,

and identify potential delays. The key elements of a Gantt chart include:

- **Tasks/Activities**: Listed on the left-hand side.
- **Timeline**: Represented along the top, usually in days, weeks, or months.
- **Bars**: Represent the duration of each task and how long it will take.
- **Dependencies**: Dependencies between tasks are often shown as arrows connecting bars.

A Gantt chart is a useful tool for project scheduling because it makes it easy to visualize project timelines, allocate resources, and identify potential bottlenecks.

UNIT 3

1. What is SRS? What are the characteristics of a good SRS? Explain with examples (hospital management system, college management).

- **SRS (Software Requirements Specification)** is a detailed document that describes the functional and non-functional requirements of a software system. It acts as a contract between the client and the development team, ensuring both parties agree on what the software should do and how it should behave.

Characteristics of a good SRS:

- **Clear and Concise**: Should be easy to understand without ambiguity.
- **Complete**: Includes all necessary information for system development.
- **Consistent**: Should not contain contradictions.
- **Verifiable**: Each requirement should be testable.
- **Traceable**: Should be easy to track to design, code, and tests.
- **Modifiable**: Easy to update and maintain.

Example (Hospital Management System):

- **Functional Requirements**: The system should allow a doctor to schedule appointments, view patient history, and generate prescriptions.
- **Non-functional Requirements**: The system should support 1000 concurrent users and respond to requests in less than 2 seconds.

Example (College Management System):

- **Functional Requirements**: The system should allow students to register for courses, view grades, and communicate with instructors.
- **Non-functional Requirements**: The system should be available 24/7 with an uptime of 99.9%.

2. List and explain requirement engineering tasks (or requirement engineering process).

Requirement Engineering is the process of defining, documenting, and maintaining requirements throughout the project lifecycle. The tasks involved include:

1. **Feasibility Study**: Analyze the feasibility of the project from a technical, operational, and financial standpoint.
2. **Requirements Elicitation**: Gather requirements from stakeholders through interviews, surveys, observations, and document analysis.
3. **Requirements Analysis**: Analyze and refine the gathered requirements to ensure they are clear, complete, and correct.
4. **Requirements Specification**: Document the requirements in an SRS or other formalized documents.
5. **Requirements Validation**: Ensure the documented requirements meet the needs of the stakeholders and are feasible for development.
6. **Requirements Management**: Continuously manage and track requirements as they change during the project lifecycle.

3. What is relationship? Explain Cardinality and Modality with examples.

- **Relationship** in the context of databases or modeling refers to how entities (e.g., classes, tables) are related to each other.

Cardinality refers to the number of instances of one entity that can be associated with instances of another entity.

- **Example (1:1)**: A hospital has one receptionist per shift. This is a one-to-one relationship.
- **Example (1:N)**: A professor teaches many courses. This is a one-to-many relationship.
- **Example (M:N)**: A student can enroll in many courses, and each course can have many students. This is a many-to-many relationship.

Modality refers to the minimum number of instances required in a relationship. For instance:

- **Example (Optional)**: A person can have zero or more children (zero is allowed).
- **Example (Mandatory)**: An employee must be assigned to at least one department.

4. Explain control flow model with example (state chart).

- **Control Flow Model** represents the flow of control within a system or process. It shows how different states of the system transition based on events or conditions.

State Chart (State Machine Diagram): A state chart is used to model the state of an object or system over time. It shows different states an entity can be in, the events that cause transitions, and actions that happen during the transitions.

Example: In an **Online Order System**, the states could be:

- **Ordered**: The user places an order.
- **Processing**: The order is being processed.
- **Shipped**: The order is shipped.
- **Delivered**: The customer receives the order.

Transitions occur between these states based on events like "payment received" or "shipped."

5. Explain use case with example (library management system).

- **Use Case**: A use case describes a specific interaction between a user (actor) and the system to achieve a goal.

Example (Library Management System):

- **Use Case**: Borrow Book
    - **Actor**: Student
    - **Precondition**: The student must be registered in the library system.
    - **Main Flow**:
        1. Student searches for a book in the catalog.
        2. Student selects the book.
        3. System checks the availability.
        4. If the book is available, the system issues the book to the student.
    - **Post-condition**: The book is issued, and the student's borrowing history is updated.

6. Explain activity and swim lane with example (billing counter in shopping mall).

- **Activity Diagram**: Represents the workflow or the activities involved in a process.

Swimlane Diagram: A variation of the activity diagram where activities are divided into "lanes," each representing a different actor or system component.

Example (Billing Counter in Shopping Mall):

- **Activities**: Customer selects items → Customer proceeds to counter → Cashier scans items → Cashier calculates total → Customer makes payment → Receipt is issued.
- **Swimlanes**:
    - Lane 1: **Customer** (selects items, proceeds to counter, makes payment).
    - Lane 2: **Cashier** (scans items, calculates total, issues receipt).

7. Explain DFD (Data Flow Diagram) with example.

- **Data Flow Diagram (DFD)** is a graphical representation of data flow within a system. It shows how data is processed by a system in terms of inputs and outputs.

Example (Library Management System):

- **Level 0 DFD**: The system receives user requests (borrow books, return books), processes them, and provides responses (book issued, book returned).
- **Level 1 DFD**: Further decomposes the processes, showing sub-processes like "Check book availability" and "Update library records."

In the DFD, processes are represented by circles, data stores by open-ended rectangles, and data flows by arrows.

8. Explain class diagram of library management system.

- A **Class Diagram** is a type of UML diagram that represents the structure of a system by showing its classes, attributes, methods, and the relationships between them.

Example (Library Management System):

- **Classes**:
  - **Book** (Attributes: title, author, ISBN; Methods: checkAvailability(), reserveBook())
  - **Member** (Attributes: name, membershipID; Methods: borrowBook(), returnBook())
  - **Library** (Attributes: list of books, list of members; Methods: issueBook(), returnBook())
- **Relationships**:
  - A **Library** contains many **Books** and many **Members**.
  - A **Member** borrows **Books**.

9. Repeat Question 1.

(See response to **Question 1** above).

10. Repeat Question 2.

(See response to **Question 2** above).

11. Repeat Question 3.

(See response to **Question 3** above).

12. Repeat Question 4.

(See response to **Question 4** above).

13. Repeat Question 5.

(See response to **Question 5** above).

14. Repeat Question 6.

(See response to **Question 6** above).

UNIT 4

1. What are different design concepts? Explain in details each.

Design Concepts in software engineering guide the development of a software system, ensuring it is efficient, maintainable, and adaptable. Some key design concepts include:

1. **Abstraction**: This involves simplifying complex systems by focusing on the essential characteristics and ignoring unnecessary details. Abstraction helps in managing complexity by providing higher-level views of the system.
    - **Example**: A class that represents a car might abstract away the complex details of how the engine works, exposing only a few key functions like "start" or "stop".
2. **Modularity**: Breaking down a system into smaller, self-contained modules that can be developed, tested, and maintained independently. This concept promotes reusability and maintainability.
    - **Example**: A software system for an online store may have modules like inventory management, order processing, and payment processing.
3. **Encapsulation**: The bundling of data and methods that operate on that data into a single unit, like a class. This helps in controlling access to data and protects it from unwanted modification.
    - **Example**: A bank account class might encapsulate balance and methods like "deposit" and "withdraw" to control how the balance is modified.
4. **Separation of Concerns**: This design principle suggests that different parts of a software system should address different

concerns. For example, the user interface should be separate from the business logic.

5. **Information Hiding**: Only essential details should be exposed to the user or other systems. Implementation details are hidden to prevent unnecessary dependencies.
6. **Reusability**: Designing components so that they can be used in different contexts or projects. This is often achieved by creating modular, independent parts of the system that are loosely coupled.

2. Define coupling and cohesion. Explain different types of it.

- **Coupling** refers to the degree of dependence between two modules or components of a system. It measures how tightly or loosely the modules are connected.
    - **Types of Coupling**:
        1. **Content Coupling**: One module directly modifies or relies on the internal data of another module.
        2. **Common Coupling**: Modules share access to global data.
        3. **Control Coupling**: One module controls the behavior of another by passing control information.
        4. **Stamp Coupling**: One module passes a data structure to another but only uses part of it.
        5. **Data Coupling**: Modules exchange data but do not rely on each other's internal workings.
        6. **Message Coupling**: The modules interact via messaging and are the least dependent on each other.

- **Cohesion** refers to the degree to which elements within a single module or component are related to each other. High cohesion means that the components within a module are closely related in functionality.
    - **Types of Cohesion**:
        1. **Functional Cohesion**: When all the elements of a module are working together to perform a single, well-defined task.
        2. **Sequential Cohesion**: The output of one part of the module is the input to another part.
        3. **Communicational Cohesion**: The module performs tasks related to the same data set.
        4. **Procedural Cohesion**: The elements of the module are grouped because they follow a certain sequence of execution, even if they perform different tasks.
        5. **Temporal Cohesion**: Elements are grouped because they are executed at the same time (e.g., initialization tasks).
        6. **Logical Cohesion**: Elements are grouped together because they perform similar functions but may not be related in a strict sense (e.g., a module handling both printing and logging).
        7. **Coincidental Cohesion**: Elements have no meaningful relationship to each other and are grouped together by mere coincidence.

3. Explain difference between coupling and cohesion.

- **Coupling** and **Cohesion** are two fundamental concepts in software design that measure different aspects of system structure:
    - **Coupling** refers to the interdependence between modules. Low coupling (loose coupling) is generally preferred, as it means modules are independent, making the system easier to maintain and extend.
    - **Cohesion** refers to how closely the activities within a module are related. High cohesion is desirable, as it means that the module is focused on a specific task, making it easier to understand, maintain, and reuse.

In short:

- **Coupling**: "How much one module knows or depends on another."
- **Cohesion**: "How well the tasks within a module fit together."

4. Compare procedure-oriented design and function-oriented design.

- **Procedure-Oriented Design** focuses on the sequence of actions or procedures that need to be executed. It is based on the concept of procedures (or functions) that manipulate data.
    - **Key Features**: Data is typically shared across multiple procedures, and the design follows a step-by-step approach to solving the problem.
    - **Example**: In an old-fashioned programming language like C, programs are built around functions like main(), getInput(), processData(), and displayOutput().

- **Function-Oriented Design** focuses on breaking the system into smaller functions based on the services it provides. It emphasizes reusable functions that operate on data.
    - **Key Features**: It emphasizes defining clear interfaces for functions and ensuring that the system's design aligns with the major functions of the system.
    - **Example**: A restaurant management system may include functions like addOrder(), processPayment(), and generateReceipt(), focusing on distinct operations rather than just data.

5. What is User Interface? Explain design issues while designing user interface.

- **User Interface (UI)** is the point of interaction between the user and a software system. It includes elements like buttons, menus, forms, and other graphical components that allow users to communicate with the system.

Design Issues in UI:

1. **Usability**: Ensuring the interface is intuitive and easy to use. This includes using clear labeling, logical layouts, and efficient navigation.
2. **Consistency**: Ensuring that UI elements behave in the same way across the system.
3. **Accessibility**: Making sure that the system is usable by people with disabilities (e.g., color blindness, hearing impairments).
4. **Responsiveness**: The UI should adapt to different screen sizes, devices, or orientations.

5. **Aesthetics**: The design should be visually appealing and professional, balancing usability with attractive design.
6. **Error Handling**: The UI should provide helpful error messages and guides to users when something goes wrong.

6. Explain design rules (Golden rules) for UI.

The **Golden Rules** of UI design are principles that guide the creation of a user-friendly interface:

1. **Consistency**: Design should be consistent throughout the system so that users can apply previous knowledge to new interactions.
2. **Provide Feedback**: Users should be informed about the results of their actions through visual or auditory cues.
3. **User Control and Freedom**: Users should feel in control of the system. They should be able to undo actions and make choices freely.
4. **Error Prevention**: The design should help prevent errors before they occur, through validation and clear instructions.
5. **Simple and Intuitive**: The interface should be easy to learn and use, with minimal cognitive load for the user.
6. **Minimize User Memory Load**: The interface should be designed so that users don't have to remember information from one screen to the next.

7. Repeat Question 1 (What are different design concepts? Explain in details each).

- (See explanation of **Design Concepts** above in Question 1.)

8. What is architectural style? Explain each in detail.

Architectural Style refers to the overall structure and organization of a software system. It defines the high-level components of the system and how they interact with each other.

Common Architectural Styles:

1. **Layered Architecture**: The system is divided into layers, each of which performs a specific role. Common layers include presentation, business logic, and data access.
    - **Example**: A typical web application might use a three-layer architecture with a presentation layer (UI), logic layer (business rules), and data layer (database access).
2. **Client-Server Architecture**: This style separates the system into clients and servers, where clients request services, and servers provide them.
    - **Example**: A web application where the web browser (client) sends requests to the web server for data or services.
3. **Microservices Architecture**: A system is divided into small, independent services that communicate with each other through well-defined APIs.
    - **Example**: An e-commerce website might have separate microservices for inventory management, payment processing, and order tracking.
4. **Event-Driven Architecture**: The system reacts to events that occur within it, making it suitable for real-time applications.
    - **Example**: An online auction system where the system reacts to bids in real-time.

5. **Service-Oriented Architecture (SOA)**: A design style where software components, or services, are loosely coupled and interact over a network.
6. **Component-Based Architecture**: The system is made up of components that can be developed, tested, and deployed independently.

UNIT - 5

1. Explain: Unit Testing, Cyclomatic Complexity

Unit Testing:

Unit testing is a software testing method where individual components or functions of a software system (called "units") are tested in isolation. The goal is to validate that each unit performs as expected.

- **Performed by:** Developers
- **Scope:** A single function, method, or class
- **Tools:** JUnit (Java), NUnit (.NET), PyTest (Python), etc.

Cyclomatic Complexity:

Cyclomatic Complexity is a metric used to measure the complexity of a program by quantifying the number of linearly independent paths through its source code. It helps determine the minimum number of test cases needed to achieve full branch coverage.

2. How Unit Testing Strategy Works on a Software Module? What Errors Are Commonly Found During Unit Testing?

How It Works:

1. **Identify Units**: Break down the code into individual functions or methods.
2. **Write Test Cases**: Create test inputs with expected outputs.
3. **Isolate Units**: Use stubs or mocks to isolate dependencies.
4. **Run Tests**: Execute test cases using a testing framework.
5. **Analyze Results**: Check if outputs match expected results.

Common Errors Found During Unit Testing:

- Incorrect logic or calculation errors
- Boundary condition failures

- Initialization errors (e.g., null references)
- Invalid function inputs
- Data type mismatch or casting errors
- Loop or conditional logic mistakes

3. What is Cyclomatic Complexity? Define Steps to Find Cyclomatic Complexity Using Flow Graph.

Definition:

Cyclomatic complexity measures the number of independent paths in a program. It helps in identifying the complexity of the code and guides testing.

Steps to Calculate Cyclomatic Complexity:

1. **Draw the Control Flow Graph (CFG):**
    - Each node represents a block of code.
    - Arrows (edges) represent control flow between nodes.
2. **Count:**
    - E = Number of edges (transitions between nodes)
    - N = Number of nodes (code blocks)
    - P = Number of connected components (usually 1 for a single program)
3. **Apply the Formula:**

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

Example:

For a function with:

- 10 edges
- 8 nodes
- 1 connected component

V(G)=10−8+2(1)=4V(G) = 10 - 8 + 2(1) = 4V(G)=10−8+2(1)=4

So, there are **4 independent paths**.

4. Describe Coding Standards

Coding Standards:

Coding standards are a set of guidelines or best practices used to write clean, readable, and maintainable code.

Purpose:

- Ensure consistency across team members
- Improve readability and maintainability
- Reduce errors and bugs
- Facilitate code review and collaboration

Common Coding Standard Elements:

- Naming conventions (camelCase, snake_case)
- Commenting and documentation
- Indentation and spacing
- Error handling practices
- File and folder structure
- Language-specific conventions (e.g., PEP8 for Python, Google Java Style)

5. Explain Knot Count in Detail

Knot Count:

Knot count is a measure used in software engineering to count the number of *logical knots* or *intersections* in the control flow of a program. It's an indicator of code readability and structure.

Detailed Explanation:

- A **knot** occurs when the flow of control in a program crosses over itself—this typically happens with complex and tangled logic.
- High knot count means the control flow is difficult to follow, making the code harder to test, debug, and maintain.

Purpose:

- To evaluate how “structured” the control flow is.
- To promote structured programming and reduce spaghetti code.

Good Practice:

- Keep knot count low for better code readability and maintainability.
- Encourage structured programming constructs like if-else, switch, and proper loops instead of goto.

UNIT – 6

1. Define Quality for Software. List the SQA Related Activities.

Software Quality:
Software quality refers to the degree to which a software product meets the specified requirements, customer needs, and expectations. It also includes the absence of defects and the ability to perform under stated conditions.

Software Quality Assurance (SQA) Activities:

- Requirements review and analysis
- Design and code inspections
- Testing (unit, integration, system, acceptance)
- Audits (internal and external)
- Process monitoring and improvement
- Standards and procedure enforcement
- Configuration management
- Defect tracking and reporting
- Risk management

2. What do you mean by Quality Assurance? Explain Various Factors that Affect Software Quality.

Quality Assurance (QA):
QA is a set of planned and systematic activities implemented to ensure that software processes and products conform to requirements and standards.

Factors Affecting Software Quality:

- **Correctness** – Meeting functional requirements
- **Reliability** – Consistent performance over time

- **Efficiency** – Optimal use of system resources
- **Usability** – User-friendly interface and experience
- **Maintainability** – Ease of modifications and bug fixes
- **Portability** – Ability to run on different platforms
- **Security** – Protection against unauthorized access
- **Testability** – Ease with which software can be tested

3. Explain Importance of SQA.

Importance of Software Quality Assurance:

- **Reduces defects and errors** early in development
- **Improves customer satisfaction**
- **Ensures compliance** with industry standards
- **Reduces cost and time** by avoiding rework
- **Improves software maintainability**
- **Enhances team accountability and performance**

4. Explain Formal Technical Review (FTR).

Formal Technical Review (FTR):
An FTR is a structured process of evaluating software products (like design, code, or documentation) by a team to detect errors, improve quality, and ensure adherence to standards.

Steps in FTR:

- Planning and preparation
- Review meeting with roles assigned (moderator, recorder, reviewers)
- Documentation of issues

- Rework by the author
- Follow-up and closure

5. What is Software Reliability? What is the Role of Software Maintenance in Software Product?

Software Reliability:
The probability that software will function correctly without failure under specified conditions for a defined period.

Role of Software Maintenance:

- **Corrective maintenance** – Fixing bugs
- **Adaptive maintenance** – Modifying software for new environments
- **Perfective maintenance** – Enhancing features
- **Preventive maintenance** – Improving maintainability

6. Explain: Quality Control and Quality Standards like ISO 9000 and ISO 9001.

Quality Control (QC):
QC involves activities and techniques to identify defects in the final product through testing and inspection.

ISO 9000:
A family of quality management standards ensuring consistent quality in processes.

ISO 9001:
Part of ISO 9000; focuses on quality management systems and continuous improvement. Organizations get certified to ISO 9001 to show their commitment to quality.

7. Differentiate Quality Control and Quality Assurance.

Aspect	Quality Assurance (QA)	Quality Control (QC)
Focus	Process-oriented	Product-oriented
Objective	Prevent defects	Identify defects
Activities	Audits, process checklists, reviews	Testing, inspection
Responsibility	Whole team (especially QA team)	Testers and QC team
Timing	Throughout the development	After development

8. Explain McCall's Quality Factors.

McCall's Quality Factors categorize software quality into three categories:

1. **Product Operation Factors:**
    - Correctness
    - Reliability
    - Efficiency
    - Integrity
    - Usability
2. **Product Revision Factors:**
    - Maintainability
    - Flexibility
    - Testability
3. **Product Transition Factors:**

- Portability
- Reusability
- Interoperability

These help measure how well software meets user needs and how easily it can adapt to changes.

9. Explain: Reliability and Availability.

- **Reliability:**
  The probability that software will perform without failure for a given time in a specified environment.
- **Availability:**
  The degree to which software is operational and accessible when required for use.
  Availability = (Uptime) / (Uptime + Downtime)

10. Explain Version and Change Control Management.

Version Control:
A system that records changes to files over time to allow specific versions to be recalled later (e.g., Git). Helps manage code changes in collaborative environments.

Change Control:
A structured approach to managing changes in software, ensuring that any change is evaluated, approved, implemented, and tracked.

Processes Include:

- Change request initiation
- Impact analysis
- Approval/rejection

- Implementation
- Documentation and versioning

UNIT – 7

1. Short Note on Component-Based Software Engineering (CBSE)

Component-Based Software Engineering (CBSE):
CBSE is an approach to software development that emphasizes the design and construction of computer-based systems using reusable software components.

Key Points:

- Promotes **reuse** of existing components (e.g., libraries, APIs, modules).
- Components have **well-defined interfaces** and **independent functionalities**.
- Encourages **faster development**, **cost reduction**, and **high-quality software**.
- Suitable for large-scale enterprise applications.

Benefits:

- Reduced development time and cost
- Improved reliability (components are tested)
- Easier maintenance and scalability

2. Short Note on Integrated CASE Environment / CASE Tools

CASE (Computer-Aided Software Engineering) Tools:
CASE tools are software applications that support various stages of the software development life cycle (SDLC) such as planning, designing, coding, testing, and maintenance.

Integrated CASE Environment:
An integrated CASE environment combines multiple CASE tools into one platform to provide **end-to-end support** for software development.

Categories:

- **Upper CASE** – Supports early stages like requirements and design.
- **Lower CASE** – Supports later stages like implementation and testing.
- **Integrated CASE** – Supports the entire software development process.

Importance in Software Engineering:

- Enhances **productivity** and **quality**
- Ensures **consistency** and **documentation**
- Supports **automated code generation**, **error checking**, and **project management**

3. Explain Scrum Development in Detail

Scrum is an Agile framework for developing, delivering, and sustaining complex products through iterative and incremental practices.

Key Elements of Scrum:

- **Roles:**
  - **Product Owner:** Defines product features and priorities.
  - **Scrum Master:** Facilitates the process and removes impediments.
  - **Development Team:** Cross-functional team that builds the product.
- **Artifacts:**
  - **Product Backlog:** List of all desired features.
  - **Sprint Backlog:** Tasks to be completed in the current

sprint.

    - **Increment:** The working product at the end of the sprint.

- **Events:**
    - **Sprint:** Time-boxed iteration (usually 2–4 weeks).
    - **Sprint Planning:** Decide what to do in the sprint.
    - **Daily Scrum:** 15-minute stand-up meeting to check progress.
    - **Sprint Review:** Demonstrate the increment to stakeholders.
    - **Sprint Retrospective:** Reflect and improve the process.

Benefits of Scrum:

- Faster delivery of functional software
- Better adaptability to changes
- Increased collaboration and team accountability

4. Write Note: Safety Engineering, Security Engineering, Resilience Engineering

Safety Engineering:

- Ensures the software/system operates without causing **unacceptable risk** or harm to users or environment.
- Focuses on **hazard analysis**, **fault tolerance**, and **fail-safe mechanisms**.
- Common in domains like aerospace, healthcare, automotive.

Security Engineering:

- Concerned with protecting software from **unauthorized access**, **attacks**, and **data breaches**.

- Involves **authentication**, **authorization**, **encryption**, **firewalls**, etc.
- Ensures **confidentiality**, **integrity**, and **availability** of systems.

Resilience Engineering:

- Focuses on the system's ability to **adapt**, **recover**, and **continue functioning** under stress, faults, or unexpected disruptions.
- Goes beyond safety and security by ensuring **continuous operation** and **graceful degradation**.
- Often applied in **critical infrastructure systems** and **cloud computing**.

UNIT – 8

1. What are the Different Types of Software Reuse? Provide Examples.

Software reuse refers to the process of using existing software components in new applications to reduce time, cost, and effort.

Types of Software Reuse:

1. **Code Reuse:**
    - Reusing existing source code or libraries.
    - *Example:* Using a Java library like Apache Commons for string handling.
2. **Component Reuse:**
    - Using self-contained software components with defined interfaces.
    - *Example:* A payment gateway component integrated into multiple e-commerce platforms.
3. **Service Reuse (Service-Oriented Architecture - SOA):**
    - Reusing web services or microservices.
    - *Example:* Reusing a REST API for user authentication across different applications.
4. **Design Reuse:**
    - Reusing design patterns, templates, or architecture blueprints.
    - *Example:* Applying the Model-View-Controller (MVC) pattern in different projects.
5. **Requirement Reuse:**
    - Reusing requirement specifications from similar systems.
    - *Example:* A hospital management system can reuse

patient registration requirements.

6. **Test Reuse:**
    - Reusing test cases or test scripts.
    - *Example:* Regression test suites reused in different releases.

2. Challenges and Barriers to Effective Software Reuse & Mitigation Strategies

Challenges and Barriers:

- **Lack of standardization:** Inconsistent coding and design standards make reuse difficult.
- **Poor documentation:** Components are hard to understand or use without proper documentation.
- **Integration issues:** Compatibility problems between reused components and new systems.
- **Quality concerns:** Reused components may have hidden bugs or be poorly tested.
- **Cultural resistance:** Developers may prefer to "build from scratch" rather than reuse.

Mitigation Strategies:

- Establish **standard coding and documentation practices**.
- Build and maintain a **centralized reusable component repository**.
- Use **wrapper components** or **adapters** to resolve integration issues.
- Perform **thorough testing** and quality assurance on reusable assets.

- Provide **training and incentives** to promote a reuse-friendly culture.

3. Key Requirements and Constraints of Real-Time Software Systems

Key Requirements:

- **Timeliness:** Meet strict deadlines (hard or soft real-time).
- **Reliability:** Must operate without failure, especially in critical systems.
- **Determinism:** Predictable behavior under all conditions.
- **Concurrency:** Handle multiple tasks/events simultaneously.
- **Minimal Latency:** Fast response to inputs or changes in environment.

Constraints:

- **Resource limitations:** CPU, memory, bandwidth may be limited.
- **Hardware dependency:** Tightly coupled with specific devices or sensors.
- **Complex timing and scheduling:** Requires real-time operating systems (RTOS).
- **High cost of failure:** Failure can cause physical harm or financial loss.

Design and Implementation Solutions:

- Use **Real-Time Operating Systems (RTOS)** like VxWorks or FreeRTOS.
- Apply **priority-based scheduling** algorithms (Rate Monotonic, EDF).
- Use **formal methods** and **timing analysis tools**.

- Optimize **interrupt handling** and **task execution time**.
- Modular and **fault-tolerant design** for recovery and reliability.

4. What is Systems Engineering? How Does it Differ from Software Engineering?

Systems Engineering (SE):
An interdisciplinary field that focuses on the **design, integration, and management of complex systems** over their life cycle, including hardware, software, people, processes, and infrastructure.

Software Engineering (SWE):
A sub-discipline of SE, focusing solely on the **systematic development and maintenance of software applications**.

Key Differences:

Feature	Systems Engineering	Software Engineering
Scope	Entire system (hardware, software, etc.)	Only software
Focus	Integration, interfaces, system lifecycle	Code quality, performance, maintainability
Tools	SysML, MBSE, DOORS	UML, Git, JIRA
Disciplines involved	Multidisciplinary	Primarily software-specific

5. Key Principles and Processes of Systems Engineering + Models & Frameworks

Key Principles:

- **Holistic Thinking:** Consider the system as a whole, including all subsystems.

- **Lifecycle Orientation:** Covers the entire lifecycle — from requirements to disposal.
- **Iterative Development:** Continuous refinement and validation.
- **Interdisciplinary Collaboration:** Involves multiple engineering disciplines.
- **Requirement-Driven:** Clearly defined and traceable requirements.

Key Processes:

1. **Requirements Engineering** – Gather and analyze stakeholder needs.
2. **System Architecture Design** – Define the structure and interactions of components.
3. **Modeling and Simulation** – Test system behavior before implementation.
4. **Verification and Validation (V&V)** – Ensure the system meets specs and user needs.
5. **Integration and Testing** – Combine components and test system functionality.
6. **Deployment and Maintenance** – Deliver and support the system post-deployment.

Models and Frameworks:

- **V-Model:** Emphasizes verification and validation at each development stage.
- **INCOSE SE Framework:** Guidelines by the International Council on Systems Engineering.
- **Model-Based Systems Engineering (MBSE):** Uses models (like SysML) instead of documents.
- **Waterfall for Systems Engineering:** Used in large, predictable

projects (e.g., aerospace).

- **Spiral Model:** Combines iterative development with risk assessment.