UNIT-1

Q1. What is Software Engineering?

- It is the **systematic approach** to developing software.
- Uses **engineering principles** to ensure software is reliable and efficient.
- Covers the **entire software lifecycle**: requirements, design, development, testing, deployment, and maintenance.

 # Objectives of Software Engineering:

- **Develop high-quality software** that meets user needs.
- Ensure **reliability** and **performance**.
- Allow **easy maintenance** and future updates.
- Ensure **cost-effective** and **on-time delivery**.
- Support **scalability** for future growth.
- Encourage **team collaboration** and organized development.

Q2. Problems Faced in Software Engineering

1. Changing Requirements

- Client needs may change frequently.
- Causes rework and delays.

2. Time and Budget Constraints

- Projects often exceed estimated time or cost.
- Leads to compromised quality or features.

3. Poor Communication

- Misunderstanding between clients and developers.
- Results in incorrect or incomplete software.

4. Complexity

- Large systems are hard to understand and manage.
- Increases bugs and errors.

5. Quality Assurance

- Difficult to detect all bugs.
- Incomplete testing leads to faults in the final product.

6. Maintenance

- Updating old software is hard, especially if poorly documented.
- Can be more costly than initial development.

7. Lack of Proper Tools

- Using outdated or wrong tools reduces productivity.
- Affects the quality of the final software.

8. Security Issues

- Software is vulnerable to cyber attacks.
- Protecting user data is difficult but essential.

Q3. **Software Engineering as a Layered Technology :-**

Software Engineering is called a **layered technology** because it is built on a **framework of interdependent layers**, where each layer supports the one above it. This layered structure ensures a **systematic approach** to software development.

🔹 **1. Quality Focus (Core Layer)**

- The **foundation layer** of software engineering.
- Ensures the delivery of **high-quality software**.
- Focuses on **reliability**, **efficiency**, **maintainability**, and **usability**.
- Acts as a guiding principle across all other layers.
-

🔹 **2. Process Layer**

- Defines the **framework** for software development.
- Describes the **software life cycle** and activities involved.
- Ensures **project management**, **planning**, and **control**.
- Common process models:
    - **Waterfall Model**
    - **Spiral Model**
    - **Agile Methodology**
    - **DevOps**

🔹 **3. Methods Layer**

- Provides **technical methods** for software development.
- Covers activities like:
  - **Requirement analysis**
  - **Software design**
  - **Coding**
  - **Testing**
- Uses tools like **UML diagrams**, **flowcharts**, and **DFDs (Data Flow Diagrams)**.

🔹 **Tools Layer**

- Supports the process and methods using **automated or semi-automated tools**.
- Improves **productivity**, **accuracy**, and **efficiency**.
- Common tools include:
  - **IDEs (e.g., Eclipse, Visual Studio)**
  - **Version Control Systems (e.g., Git)**
  - **Testing Tools (e.g., Selenium, JUnit)**
  - **Project Management Tools (e.g., JIRA)**

🔹 **4. Interdependency of Layers**

- Each layer depends on the **support and structure** of the lower layers.
- Quality is influenced by the strength of processes, methods, and tools used.
- Ensures **consistency**, **scalability**, and **structured development**.

✅ **Conclusion**

- The layered approach in software engineering ensures a **disciplined, repeatable, and manageable** way to build software.
- It enhances **software quality**, improves **team coordination**, and reduces **development risks**.

Q4. Software Components –

- A **software component** is a **self-contained, independent, and reusable** unit of software that performs a specific function within a larger system.
- It consists of **interface** and **implementation**:
    - **Interface**: Defines how other components interact with it.
    - **Implementation**: The internal logic or code of the component.

 # Key Elements of a Software Component

1. **Component Interface**
    - Exposes **public methods**, **properties**, and **events**.
    - Defines **communication** with other components.
2. **Component Implementation**
    - Contains **core code** (business logic, data processing).
    - Hidden to **support encapsulation**.
3. **Component Metadata**
    - Provides **version**, **dependencies**, and **documentation**.

 # Types of Software Components

1. **Presentation Components (UI Layer)**
    - Manages **user interaction** and **display**.
    - Example: Login page, menu bar.
2. **Business Logic Components**
    - Handles **core processing** and **decision-making** tasks.
    - Example: Tax computation, order processing.
3. **Data Access Components**
    - Manages **database** interactions.
    - Example: JDBC, DAO.
4. **Utility Components**
    - Provides **general-purpose functionalities**.
    - Example: Logging, encryption utilities.
5. **Middleware Components**

- Facilitates **communication** between distributed components.
- Example: CORBA, COM, RMI.

Q5. Main Characteristics of Good Software / Software Quality Attributes :-

 # Software Quality Attributes –

➔ these are the **non-functional requirements** that describe **how well the software performs**, rather than what it does.
➔ this help **evaluate the overall quality** of the system.

1. Correctness (Functionality)
   → Software meets **user requirements and performs intended tasks**.
   → **Accurate output** based on given inputs.
2. Reliability
   → Operates consistently without failures or crashes.
   → Handles errors and continues to function smoothly.
3. Usability
   → Software is easy to learn and use.
   → Intuitive interface with simple navigation.
4. Efficiency
   → Uses minimal resources (memory, CPU, etc.) for fast performance.
   → Optimized speed even under heavy load.
5. Maintainability
   → Software is easy to update and fix.
   → Well-documented code for easier debugging and enhancements.
6. Portability
   → Can run on multiple platforms (Windows, Linux, etc.) with little modification.
   → Adapts easily to different environments.
7. Security
   → Protects data from unauthorized access and attacks.
   → Uses encryption and authentication for safety.
8. Scalability
   → Handles increased demand efficiently (both horizontally and vertically).
   → Grows without sacrificing performance.

9. Flexibility
   ➔ Easily adapts to changing requirements.
   ➔ Supports new features and upgrades.
10. Safety
   ➔ Ensures safe operation, especially for critical applications (e.g., medical, aerospace).
   ➔ Minimizes risks during software execution.

Q6. **Software Crisis -**

Main Reasons:

1. **Complexity** of modern software systems.
2. **No standard methods** in development.
3. **Fast tech changes** make it hard to keep up.
4. **Poor planning** and **management** of projects.
5. **Too many demands** for new software and features.
6. **Difficult to estimate** time and costs accurately.

Main Results:

1. **Poor quality software** with bugs and errors.
2. **Exceeded costs and deadlines**.
3. **Hard to maintain** and update software.
4. **Frustration** for developers and users.
5. **Loss of trust** in the software development process.
6. **Late delivery of software**

Q7. **What is Software Quality & three dimensions of it ?**

1. It means the software **works correctly** and **meets user needs**.
2. It should perform all tasks as **expected and required**.
3. The software must be **reliable**, **fast**, and **secure**.
4. It should be **easy to fix**, **update**, and **maintain**.
5. Good quality software has **fewer bugs** and is **user-friendly**.
6. Overall, it should be **fit for use** in real-world conditions.

Three Dimensions of Software Quality :-

1. Quality of Design
    - How well the software is planned to meet user needs and requirements.
    - Focuses on architecture and features.
    - It ensures a strong foundation.
2. Quality of Conformance
    - How closely the software follows the design specifications.
    - Ensures the software is built correctly with no defects.
    - It ensures the software matches that foundation.
3. Quality of Use
    - How well the software performs in real-world conditions.
    - Ensures the software is user-friendly, reliable, and meets expectations.
    - It ensures the software works well in practice, meeting user needs.

Q8. What is Software Development Life Cycle (SDLC)?

➔ It stands for **Software Development Life Cycle**.
➔ It is a **structured approach** for **developing** software, from initial planning to maintenance.
➔ Ensures **high-quality software** is delivered on time and within budget.
➔ Helps in **systematic planning**, **design**, **development**, **testing**, and **deployment** of software.
➔ It is a pictorial & diagrammatic representation of the software life cycle.

 # Stages of SDLC :-

1. **Planning**

➔ Define the **project goals** and plan the entire process.
➔ **Key Activities**:

- Set **scope** (what the software will do).
- Define **resources**, **budget**, and **timeline**.

- Identify **project risks** and create a **risk management plan**.

2. **Analysis**

➔ Gather detailed **requirements** from the users.
➔ **Key Activities**:

- Collect **functional requirements** (what the software should do).
- Identify **non-functional requirements** (how the software should perform).
- Create a **requirement specification document** to guide development.

3. **Design**

➔ Plan how the software will be **structured** and built.
➔ **Key Activities**:

- Create **system architecture** (high-level design).
- Plan **database** and **module design** (low-level design).
- Define how **modules** will interact with each other.

4. **Development (Coding)**

➔ **Convert design into working code**.
➔ **Key Activities**:

- Write the **source code** based on design documents.
- Follow **coding standards** and guidelines.
- Conduct **unit testing** to check for coding errors.

5. **Testing**

➔ Ensure the software works **correctly** and has **no bugs**.
➔ **Key Activities**:

- Perform **unit testing** (testing individual parts).
- Perform **integration testing** (testing combined components).
- Conduct **user acceptance testing (UAT)** with real users to validate the software.

6. **Deployment**

➔ **Release** the software for users to use.
➔ K**ey Activities**:

- **Install** the software on user systems or servers.
- Ensure everything works in the **live environment**.
- Provide **training** and **documentation** for users.

7. **Maintenance**

➔ **Keep the software running smoothly** after deployment.
➔ **Key Activities**:

- **Fix bugs** reported by users.
- Add new **features** or **improvements** based on feedback.
- Perform regular **updates** to maintain software quality.

Q9. **Prototype Model of SDLC –**

➔ It is a software development method where a **working model (prototype)** of the system is quickly built, shown to the user, and **repeatedly improved** based on feedback.
➔ This helps in **understanding unclear requirements** early in the process.

 # Key Steps of Prototype Model :

1. **Requirements Gathering**
   - Collect basic and unclear requirements from the user.
   - Focus is on **what the user wants**, not full details.
2. **Quick Design**
   - Create a **rough design** with main features and user interface.
3. **Build Prototype**
   - Develop a **sample version** (mock-up) of the software.
   - May not have full functionality.
4. **Customer Evaluation of Prototype**
   - Show the prototype to the user.
   - Get **feedback on features, flow, and design**.
5. **Refine Requirements**
   - Use feedback to **modify and improve** the design.
   - Repeat steps 2 → 3 → 4 → 5 until the user is satisfied.
6. **Design**
   - Create a **final detailed design** of the actual system.
7. **Implementation**
   - Write the full code to develop the complete system.
8. **Testing**

- Test the software for errors and functionality.

9. **Maintenance**
    - Fix bugs, update features, and support the system after delivery.

⭐ **Advantages Over Conventional Models (like Waterfall):**

- Handles **unclear or changing requirements** better.
- **Customer feedback** is included from the beginning.
- Reduces the **risk of failure**.
- Saves **time and cost** in the long run.
- Helps **visualize the final product** early.

✅ **Advantages:**

- Users understand the system early.
- Better communication with users.
- Errors found early.
- More flexible than Waterfall.

⚠️ **Disadvantages:**

- Takes more time due to changes.
- Users may think prototype is final.
- Frequent changes can increase cost.

Q10. I**terative Waterfall Model –**

➔ It is an improved version of the traditional **Waterfall Model**.
➔ Unlike the original, it allows **feedback and revisions** between stages.
➔ Each phase flows to the next like a waterfall, but **iterations (repeats)** are possible if errors or changes are needed.

 # Phases and Activities

1. **Requirement Analysis**
    - Understand what the user wants.
    - Collect and document all system requirements.
    - Output: **Requirement Specification Document**.

2. **Design**
    - Plan the architecture of the system.
    - Divide into modules, define input/output, and database design.
    - Output: **System Design Document**.

3. **Implementation (Coding)**
    - Developers write code based on the design.
    - Use programming languages and tools.
    - Output: **Working software modules**.

4. **Testing**
    - Test software to find and fix bugs.
    - Unit testing, integration testing, and system testing.
    - Output: **Bug-free software**.

5. **Deployment & Maintenance**
    - Deploy the software to the user.
    - Fix issues and update software as needed.
    - Output: **Maintained and updated software**.

Q12. **Spiral Model of SDLC –**

➔ It is a **risk-driven software development process** that combines the features of the **Waterfall** and **Prototyping models**.

➔ The Spiral Model is a flexible and iterative approach to software development that combines elements of both the Waterfall and Prototype models.

➔ It follows **repeated cycles (spirals)** where each loop represents a **phase in development**.

 # Phases of Spiral Model:-

Each spiral consists of **four key phases**:

1. Planning Phase

- Collect requirements from the customer.

- Define objectives and alternatives for the next development cycle.
- Identify constraints (budget, time, technology).

Output: Requirements & Planning Documents.

2. Risk Analysis Phase

- **Identify possible risks** in technology, cost, time, performance, etc.
- Analyze how to **reduce or avoid** these risks.
- Create a **prototype** if needed to reduce uncertainty.

📌 **Why important- RISK ANALYSIS :**

- This model is **centered around risk management**
- **Each loop starts with Risk Analysis**, making sure **problems are solved before development**.
- It gives a **flexible and safe approach** for complex, high-budget, and uncertain projects.
- Helps in **early detection of major issues**, reducing cost of late-stage errors.

Output: Risk resolution plan & updated prototype (if needed).

3. Engineering Phase

- Actual **design, coding, and testing** of the software takes place.
- Build version based on planned requirements and resolved risks.

Output: Working software version.

4. Evaluation Phase

- Customer evaluates the software output.
- Feedback is collected for the next cycle.
- Decide whether to continue, modify, or stop development.

Output: Approval to proceed to the next spiral.

 # How Spiral Model Works:

- The model is repeated in **loops**, each producing a **more refined version** of the software.
- **Each spiral loop = one complete SDLC cycle** (Plan → Risk → Build → Review).

 # Advantages :-

- Best suited for **large and high-risk projects**.
- Risks are handled **early and carefully**.
- Customer feedback is involved at every stage.

Q11. How Waterfall & Prototype Models Are Accommodated in the Spiral Model :

➔ The Spiral Model effectively combines the strengths of both the **Waterfall Model** and the **Prototype Model** to provide a flexible, risk-driven software development process.
➔ It uses the **structured, phase-wise approach** of the Waterfall Model for stable and well-defined parts of the project, ensuring discipline and clarity.
➔ Simultaneously, it incorporates **prototyping techniques** to handle uncertain, high-risk areas by enabling early user feedback and iterative refinement.
➔ By integrating these two models within its iterative spirals, the Spiral Model achieves a **balance between predictability and flexibility**, making it ideal for complex and large-scale software projects.
➔ It reduces risks, adapts to changing requirements, and ensures better system quality through continuous evaluation and improvement.

 # List Some Software Process Paradigms :-

1) Procedural Paradigms
2) Data Driven Paradigms
3) Object oriented Paradigms

Q13. Evolutionary Development Model (EDM)

- The Evolutionary Development Model is a **software development approach** where the system is developed **incrementally**, allowing it to evolve over time.
- It focuses on **building an initial version quickly**, then **improving it through multiple iterations** based on user feedback.
- **Incremental delivery** of software.
- **Continuous feedback** from users after each iteration.
- **Overlapping phases** (requirements, design, coding, testing).
- Helps in handling **changing or unclear requirements**.
- Often used in **Agile** and **DevOps** environments.

 # Process Phases:

1. **Initial Requirement Gathering:**
    - Collect basic and core requirements.

- Not all requirements need to be fully defined initially.

2. **Initial System Development:**
   - A basic working version (core functionalities) is developed.
3. **User Evaluation:**
   - The user tests the version and provides feedback.
   - Feedback includes suggestions, errors, and new requirements.
4. **Refinement and Enhancement:**
   - Developers update the system by adding more features or correcting issues.
   - The process repeats in several **evolutionary cycles**.
5. **Final System Delivery:**
   - After several iterations, a full and final system is delivered.

Q14. Which is More Important: Product or Process? – Summary in Points

🔹 **1. Product:**

- The **final software** delivered to the customer.
- Includes features, performance, UI, and documentation.
- Directly impacts **user satisfaction and business value**.
- A good product is the **main goal** of software engineering.

🔹 **2. Process:**

- The **methodology or approach** used to build the product.
- Includes **planning, design, coding, testing, and maintenance**.
- Ensures **efficiency, quality, and risk management**.
- A good process results in **consistent and maintainable** products.

🔹 **3. Why Product is Important:**

- Final output that meets **customer requirements**.
- Affects **company reputation and success**.
- Used for **performance evaluation** and feedback.

🔹 **4. Why Process is Important:**

- Defines **how efficiently and effectively** a product is built.
- Reduces **errors, rework, and development cost**.
- Ensures **repeatability and continuous improvement**.
- Enables better handling of **complex and large projects**.

🔹 **5. Conclusion:**

- Both are important and **complement each other**.
- A **good product needs a good process** behind it.
- In the **long term, the process is slightly more important** because it ensures quality, reliability, and continuous improvement of the product.

Q15. McCall's Quality Factors with Quality Triangle :

- McCall's quality model (developed in 1977) is one of the earliest models to define **software quality**.
- It defines quality based on the **needs of users, developers, and maintainers**.
- Give **structured evaluation** of software quality.
- Used as a **foundation for modern quality models**.
- The model organizes quality into **three major perspectives**, forming a **Quality Triangle**:
    - **Product Operation**
    - **Product Revision**
    - **Product Transition**

 # Quality Triangle – Three Main Perspectives:

A. Product Operation (During software use):

1. **Correctness** – Performs all required functions accurately.
2. **Reliability** – Works consistently without failure.
3. **Efficiency** – Uses system resources optimally (CPU, memory, time).
4. **Integrity** – Protects data from unauthorized access.
5. **Usability** – Easy to learn and operate for users.

B. Product Revision (During maintenance and updates):

6. **Maintainability** – Easy to find and fix issues.

7. **Flexibility** – Easy to modify or enhance for new needs.
8. **Testability** – Easy to test for bugs and errors.

C. Product Transition (When moving to a new environment):

9. **Portability** – Can run on different hardware or OS.
10. **Reusability** – Code/components can be reused in other projects.
11. **Interoperability** – Can work with other systems or software.

Q16. Difference between Horizontal & Vertical Partitioning:-

Aspect	Horizontal Partitioning	Vertical Partitioning
1. Definition	Divides the system into layers based on **functional tasks**.	Divides the system based on **major functions or features**.
2. Focus	Focuses on **separating input, processing, and output** tasks.	Focuses on **separating control and processing logic** for features.
3. Structure	Layers such as **UI → Business Logic → Data Layer**.	Modules for specific tasks like **login, payment, reports**, etc.
4. Flow Direction	Flow of control or data is usually **top to bottom**.	Flow of control is usually **feature-wise (end-to-end)**.
5. Example	Web application: UI layer, logic layer, data access layer.	ATM system: Card validation, transaction processing, receipt print.

Q17. Generic Software :-

- Software designed for **wide applications**, not customized for specific tasks.
- **Ex**: Word processors, spreadsheets, email clients.
- **Reusable** across multiple users/organizations.
- **Configurable** for user-specific settings.
