Unit - II Software Engineering Requirement

✅ 1. Software Engineering Core Principles

These principles are **fundamental guidelines** to help engineers build software that is reliable, efficient, maintainable, and user-focused.

🔹 1. The Reason It All Exists

- Software is created to solve a problem or meet a specific need.
- Always focus on **adding value** to the customer or end-user.

🔹 2. KISS (Keep It Simple, Stupid)

- Simple solutions are often better and easier to maintain.
- Avoid unnecessary features and complexity.

🔹 3. Maintain the Vision

- Keep a clear goal throughout development.
- Ensure every team member understands the **core purpose** of the product.

🔹 4. What You Produce, Others Will Consume

- Write clear code and documentation that others (like future developers or testers) can understand and use.

🔹 5. Be Open to the Future

- Design software to be **scalable and adaptable**, so it can evolve over time without major rewrites.

🔹 6. Plan Ahead for Reuse

- Create modular, well-documented components that can be reused in other projects or systems.

🔹 7. Think!

- Always think before you code. Analyze the problem, assess solutions, and anticipate challenges.

🛠️ Software Engineering Practices

✅ Communication Practices

Definition: Involves the continuous exchange of **accurate and complete information** between stakeholders (clients, users, developers, testers).

Key Points:

- Poor communication leads to misunderstood requirements and project failure.
- Involves **elicitation**, **clarification**, and **validation** of requirements.

Techniques Used:

- Interviews, questionnaires, brainstorming
- Use Case Diagrams, User Stories
- Prototypes or wireframes for visual understanding

Communication Principles are:-

1. Listen: Try to focus on the speaker's words.Ask for clarification if something is unclear, but avoid constant interruptions.

2. Prepare before you communicate: Speed the time to understand the problem before you meet with others. If necessary, do some research to understand business domain jargon.

3. Someone should facilitate the activity: Every communication meeting should have a leader to keep the conversation moving in a productive direction.

4. Face-to Face communication is best: It usually works better when some other representation of the relevant information is present. For eg. A participant may create a document that serves as a focus for discussion.

5. Take notes and document decisions: Someone participating in the communication should serve as a "recorder" and write down all important points and decisions.

6. Strive for collaboration: Each small collaboration serves to build trust among team members and creates a common goal for the team.

7. Stay focused; modularize your discussion: The facilitator should keep the conversation modular; leaving one topic only after it has been resolved.

8. If something is unclear, draw a picture: A sketch or drawing can often provide clarity when words fail to do the job.

9. A) Once you agree to something, move on. **B) If you can't agree to something, move on C) If a feature or function is unclear and cannot be clarified at the moment, move on.** : Rather than iterating endlessly, the people who participate should recognize that many topics require discussion and that "moving on" is sometimes the best way to achieve communication agility.

10. Negotiation is not a contest or a game. It works best when both parties win: There are many instances in which you and other stakeholders must negotiate functions and features, priorities, and delivery dates.

✅ **Planning Practices**

Definition: Planning is about defining the **scope**, **timeline**, **cost**, and **resources** for a software project.

Objectives:

- Identify **deliverables** and set deadlines
- Allocate resources wisely
- Minimize risks by anticipating problems

Key Activities:

- Creating a project plan and schedule
- Risk identification and mitigation planning
- Cost and effort estimation (e.g., using COCOMO model)

Planning Principles:-

1. Understand the scope of the project: It's impossible to use a road map if you don't know where you are going. Scope provides the software team with a destination.

2. Involve stakeholders in the planning activity: Stakeholders define priorities and establish project constraints.
3. Recognize that the planning is iterative: When the project work begins it's likely that few things may change. To accommodate these changes the plan must be adjusted, as a consequence.
4.Estimate based on what you know: The purpose of estimation is to provide an indication of the efforts, cost, and task duration. If the information is vague or unreliable, estimates will be equally unreliable.
5.Consider the risk as you define the plan: The team should define the risks of high impact and high probability. It should also provide a contingency plan.
6.Be realistic: The realistic plan helps in completing the project on time including the inefficiencies and change.
7. Adjust granularity as you define the plan: Granularity refers to the level of details that is introduced as a project plan is developed. It is the representation of the system from macro to micro level. A "high-granularity" plan provides significant work task detail. A "low-granularity" plan provides broader work tasks that are planned over longer time periods.
8. Define how you intend to ensure quality: The plan should identify how the software team intends to ensure quality. If technical reviews are to be conducted, they should be scheduled.
9.Describe how you intend to accommodate change: Even the best planning can be obviated by uncontrolled change. The software team should identify how the changes are to be accommodated as the software engineering work proceeds. If a change is requested, the team may decide on the possibility of implementing the changes or suggest alternatives.
10. Track and monitor the plan frequently and make adjustments if required: Software projects fall behind schedule one day at a time. Therefore, make sense to track progress on a daily basis, looking for problem areas and situations in which scheduled work does not conform to actual work conducted. When shippage is encountered, the plan is adjusted accordingly.

✅ Modeling Practices

Definition: Modeling involves creating **abstract representations** of the system to better understand and plan how it should be built.

Types of Models:

- **Structural Models:** Class Diagrams, Entity-Relationship Diagrams
- **Behavioral Models:** Sequence Diagrams, Activity Diagrams

- **Functional Models:** Data Flow Diagrams (DFDs), Use Cases

Benefits:

- Identifies potential issues early
- Improves understanding among team members
- Acts as a **blueprint** for developers

Analysis Modelling Principles:

1. The information domain of a problem must be represented and understood

The information domain encompasses the data that flow into the system from end users, other systems or external devices.

2. The functions that the software performs must be defined.

Software functions provide direct benefit to end users and also provide internal support for those features that are user visible. Some functions transform data that flow into the system.

3. The behavior of the software (as a consequence of external events) must be represented)

The behavior of computer software is driven by its interaction with the external environment. Input provided by end users, control data provided by an external system, or monitoring data collected over a network all cause the software to behave in a specific way.

4. The models that depict information function and behavior must be partitioned in a manner that uncovers detail in a layered fashion.

Requirements modeling is the first step in software engineering problem solving. It allows you to better understand the problem and establishes a basis for the solution (design).

5. The analysis task should move from essential information toward implementation detail.

Requirements modeling begins by describing the problem from the end-user's perspective. The "essence" of the problem is described without any consideration of how a solution will be implemented.

Design principles are:

1. Design should be traceable to the requirements model.

The design model should translate the information into architecture; a set of subsystems which implement major functions and a set of component level designs are the realization of the analysis classes.

2. Always consider the architecture of the system to be built

Software architecture is the skeleton of the system to be built. It affects interfaces, data structures, program control flow and behavior, the manner in which testing can be conducted, and the maintainability of the resultant system.

3. Design of data is as important as design of processing functions

Data design is an essential element of architectural design. A well-structured data design helps to simplify program flow, makes the design and implementation of software components easier.

4. Interfaces (both internal and external) must be designed with care

The manner in which data flows between the components of a system has much to do with processing efficiency, error propagation, and design simplicity. A well-designed interface makes integration easier and assists the tester in validating component functions.

5. User interface design should be tuned to the needs of the end user

The user interface is the visible manifestation of the software. No matter how sophisticated its internal functions, no matter how well designed its architecture, a poor interface design often leads to the perception that the software is “bad”.

6. Component-level design should be functionally independent,

Functional independence is a measure of the “Single-mindedness” of a software component. The functionality that is delivered by a component should be cohesive-that is, it should focus on one and only one function or sub function.

7. Components should be loosely coupled to one another and to the external environment.

Coupling is achieved in many ways- via a component interface, be messaging, through global data. As the level of coupling increases, the likelihood of error propagation also

increases and the overall maintainability of the software decreases. Therefore, component coupling should be kept as low as is reasonable.

8. Design representations(models) should be easily understandable

The purpose of design is to communicate information to practioners who will generate code, to those who will test the software, and to others who may maintain the software in the future. If the design is difficult to understand, it will not serve as an effective communication medium.

9. The design should be developed iteratively. With each iteration, the designer should strive for greater simplicity

Like almost all creative activities, design occurs iteratively. The first iterations work to refine the design and correct errors, but later iterations should strive to make the design as simple as possible.

✅ Construction Practices

Definition: The actual process of **building** the software system through **coding**, followed by **testing** and **debugging**.

Key Aspects:

- Use of proper **programming practices** and **naming conventions**
- Writing **modular and reusable code**
- Applying unit testing and integration testing
- Use of version control tools like Git

Standards:

- Follow ISO/IEC coding standards
- Use test-driven development (TDD) or behavior-driven development (BDD)

CODING PRINCIPLES ARE AS FOLLOW:-

Preparation Principles:

Before you write one line of code, be sure you

1. Understand of the problem you're trying to solve
2. Understand basic design principles and concepts
3. Pick a programming language that meets the needs of the software to be built and the environment in which it will operate
4. Select a programming environment that provides tools that will make your work easier
5. Create a set of unit tests that will be applied once the component you code is completed

Coding Principles:

1. As you begin writing code, be sure you:
2. Constrain your algorithms by following structured programming practice.
3. Consider the use of pair programming
4. Select data structures that will meet the needs of the design
5. Understand the software architecture and create interfaces that are consistent with it.
6. Keep conditional logic as simple as possible.
7. Create nested loops in a way that makes the theme easily testable.
8. Select meaningful variable names and follow other local coding standards
9. Write code that is self-documenting
10. Create a visual layout that aids understanding

Validation Principles: After you've completed your first coding pass, be sure you

1. Conduct a code walkthrough when appropriate
2. Perform unit tests and correct errors you've uncovered
3. Refactor the code

✅ Software Deployment Practices

Definition: Deployment is the process of **releasing the software to users**, installing it in the target environment, and ensuring it functions correctly.

Deployment Stages:

1. **Release Preparation:** Packaging, configuration, and final testing
2. **Installation and Setup:** Installing on servers or user systems
3. **Post-Deployment Support:** Handling bug fixes, updates, and user feedback

Modern Practices:

- Use of **CI/CD (Continuous Integration/Continuous Deployment)** tools like Jenkins, GitLab
- **DevOps** practices to automate and monitor deployment

PRINCIPLES OF DEPLOYMENT:-

1. Customer expectations for the software must be managed

Before the software delivery the project team should ensure that all the requirements of the users are satisfied.

2. A complete delivery package should be assembled and tested

The system containing all executable software, support data files, tools and support documents should be provided with beta testing at the actual user side.

3. A support regime must be established before the software is delivered

This includes assigning the responsibility to the team members to provide support to the users in case of problem.

4. Appropriate instructional materials must be provided to end users

At the end of construction various documents such as technical manual, operations manual, user training manual, user reference manual should be kept ready. These documents will help in providing proper understanding and assistance to the user.

5. Buggy software should be fixed first, delivered later.

Sometimes under time pressure, the software delivers low-quality increments with a warning to the customer that bugs will be fixed in the next release.

1. Requirement Engineering

Definition: Requirement Engineering is the process of **gathering, analyzing, documenting, and validating** the needs and constraints of the stakeholders for a software system.

Its goal is to ensure that **the software meets user expectations and business needs**.

From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity. It must be adapted to the needs of the process, the project, the product, and the people doing the work.

🔄 2. Requirement Engineering Tasks

These tasks form the **backbone of the software development process**. There are typically 5 main tasks:

- **Inception**
  - Identify project stakeholders.
  - Most projects begin when a business need is identified or a potential new market or service is discovered.
  - The aim is
    - To have the basic understanding of problem
    - To know the people who will use the software
    - To know the exact nature of the problem.

- **Elicitation**

Use interviews, questionnaires, and discussions to gather basic needs.

Elicitation means to define what is required.To know the objectives of the system to be developed is a critical job.

Requirement elicitation is difficult because numbers of problems are encountered:
Problems of scope: The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.
Problems of understanding: The customers/users are not completely sure of what is needed. They don't have a full understanding of the problem domain. They omit information that is believed to be "obvious,". Specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous (unclear) or untestable.
Problems of volatility: The requirements change over time.

🔹 Elaboration (Analysis and Negotiation)

- Refine and expand requirements.
- Resolve conflicts, prioritize needs, and ensure feasibility.
- Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system.
- Each user scenario is parsed to extract analysis classes.
- The attributes of each analysis class are defined, and the services that are required by each class are identified.

🔹 Negotiation

It means discussion on financial and other commercial issues. In this step customer, user and stakeholder discuss to decode:

- To rank the requirements
- To decide priorities
- To decide risks
- To finalize the project cost
- Impact of above on cost and delivery

🔹 Specification

- Document all requirements in a structured form.

- Can be done using **Software Requirement Specification (SRS)** documents.
- A written document, a set of graphical models, a collection of scenarios, a prototype, Mathematical model.
- It serves as the foundation for subsequent software engineering activities. It describes the function, performance of a computer-based-system, constraints that will govern its development.

🔹 Validation

- Ensure that requirements are **complete, correct, and consistent**.
- Reviews, walkthroughs, and prototyping are used.
- Products are assessed for quality during the validation period.
- Provide clarification related to conflicting requirements, unrealistic expectations, etc.

🔹 Requirements Management

- Maintain and track changes to requirements throughout the project.
- Helps avoid scope creep and ensures traceability.

📑 3. Types of Requirements

Requirements are generally classified into three main types:

🔹 A. Functional Requirements

- Define **what the system should do**.
- Describe system behavior under specific conditions.
- Functional requirements describe the interaction of software with its environment and specify the inputs, outputs, external interfaces and the functions that should be included in the software.
- These requirements should be complete and consistent.
- Completeness implies that all the user requirements are defined.
- Consistency implies that all requirements are specified clearly without any contradictory definition.

- 📌 *Example:* "The system shall allow users to log in using email and password."

🔹 **B. Non-Functional Requirements**

- Define **how the system performs** tasks (quality attributes).
- Includes performance, usability, reliability, and security.
- These requirements arise due to user requirements, budget constraints, organizational policies, and so on. These requirements are not related directly to any particular function provided by the system.

Different type of Non Functional Requirement

Product Requirements:

- **Reliability Requirements** describe the acceptable failure rate of the software.
- **Usability Requirements** describe the ease with which users are able to operate.
- **Efficiency Requirements** describe the extent to which the software makes resources, the speed with which the system executes and the memory it consumes for its operation.
- **Portability Requirements** describe the ease with which the software can be transferred from one platform to another

Organizational Requirements:

(a) Implementation Requirements describe requirements such as programming language and design method.

(b) Standards Requirements describe the process standards to be used during software development. For example, ISO and IEEE standards.

(c) Delivery Requirements specify when the software and its documentation are to be delivered to the user.

External Requirements:

(a) **Interoperability Requirements** defines the way in which different computer-based systems interact with each other in one or more organizations.
(b) **Legislative Requirements** ensures that the software operates within the legal jurisdiction. For example, pirated software should not be sold.

(c) Ethical Requirements specifies the rules and regulations of the software so that they are acceptable to users.

- 📌 *Example:* "The system must handle 1000 transactions per second.

🔹 **C. Domain Requirements**

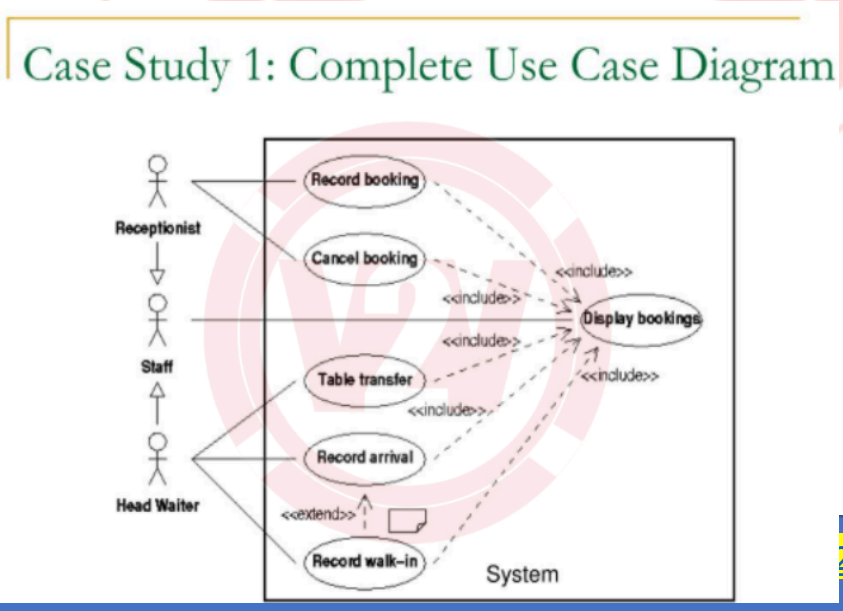
- Specific to the **domain or industry** of the software.
- Often driven by regulations, standards, or specific rules.
- 📌 *Example:* "A banking system must comply with RBI data security policies."

🎭 4. Developing Use-Cases

📘 **Definition:** A **use-case** is a written description of **how a user interacts** with the system to achieve a specific goal.
It helps in understanding **user behavior and system responses**.

🧩 **Components of a Use-Case:**

- **Use-Case Name**
- **Actor(s)** – Who interacts with the system
- **Preconditions** – What must be true before the use-case starts
- **Basic Flow** – Step-by-step normal interaction
- **Alternate Flow** – Variations in the steps
- **Exceptions** – What happens in case of errors
- **Postconditions** – Final state after use-case completes



📘 **SRS – Software Requirements Specification**

✅ 1. What is SRS? (Definition)

SRS is a **formal document** that describes the **functional and non-functional requirements** of the software system to be developed.
It acts as an **agreement** between the customer and the development team.

SRS should include both the user requirements for a system and a detailed specification of the system requirements.

Software requirement specification is a document that completely describes what the proposed software should do without describing how software will do it.

✅ 2. Need for SRS (Why SRS is Important)

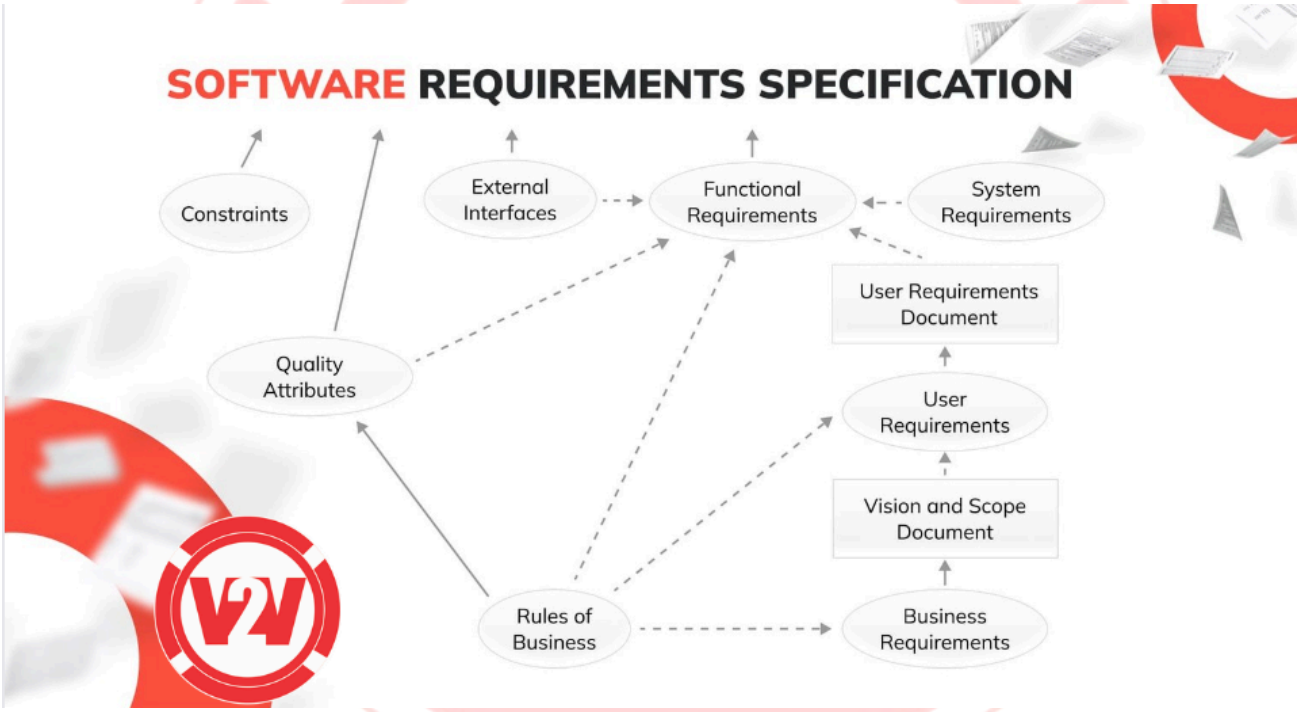
Reason	Explanation
✅ **Clear Communication**	Acts as a contract between stakeholders and developers
✅ **Requirement Clarity**	Helps avoid misunderstandings and assumptions
✅ **Project Planning**	Provides the foundation for cost, effort, and time estimation
✅ **Design & Development Base**	Used by designers and developers to build the system
✅ **Testing & Validation**	Testers use it to create test cases and validate functionality
✅ **Maintenance Reference**	Helps in future enhancements and debugging

1. An SRS is helping the clients understand their own needs or requirements.

2. An SRS is important because it establishes the basis for agreement between the client and the supplier on what the software product will do.

3. A high-quality SRS is a prerequisite to high-quality software.

4. SRS enables developers to consider user requirements before the designing of the system. This reduces rework and inconsistencies, which reduce the development efforts.
7. SRS needed to provide feedback, which ensures to the user that the organization understands the issues or problems to be solved.



Format of SRS Document

A standard SRS document follows the IEEE 830 standard. Here's a typical structure:

- Section 1: Product Overview and Summary
- Section 2: Development, Operating, and Maintenance Environments
- Section 3: External Interfaces and Data Flow
- Section 4: Functional Requirements
- Section 5: Performance Requirements
- Section 6: Exception Handling
- Section 7: Early Subsets and Implementation Priorities
- Section 8: Foreseeable Modifications and Enhancement
- Section 9: Acceptance Criteria
- Section 10: Design Hints and Guidelines
- Section 11: Cross-reference Index
- Section 12: Glossary of Terms

Characteristics of a Good SRS

Characteristic	Explanation
Correctness	SRS should describe the actual needs of the stakeholders
Completeness	It should include all significant requirements
Unambiguity	Each requirement should have only one interpretation
Consistency	No conflicting requirements or definitions
Verifiability	Each requirement must be testable (i.e., it can be verified)

Characteristic	Explanation
Modifiability	The document should be easy to update and modify
Traceability	Each requirement should be traceable to its source (client need, business rule)
Rank for Importance	All requirements are not equally important so we need to decide priority