Software Engineering & Product Management

Module 2

MODULE-2	SYSTEM ENGINEERING	22AIM51.3, 22AIM51.4	8 Hours
Requirement Engineering - Initiating the Requirements Engineering process, Eliciting Requirements, developing use cases, Building the analysis model, Software Requirement Document, System Architectural design, Subsystems development, System integration testing and deployment, System configuration management.			

---

1)Requirements Engineering

Definition:

Requirements Engineering (RE) is the systematic process of **identifying, analyzing, documenting, validating, and managing** the needs and expectations of stakeholders for a software system.
It ensures that the final product meets user requirements and business goals effectively.

1. Feasibility Study

Determines whether the proposed system is **practical and achievable**.

- **Technical Feasibility:** Checks availability of required technology, tools, and technical skills.
- **Operational Feasibility:** Ensures the system is user-friendly and maintainable.
- **Economic Feasibility:** Analyzes project **cost vs. benefits** (most important).
- **Legal Feasibility:** Ensures compliance with **laws, regulations, and IP rights**.
- **Schedule Feasibility:** Checks if the project can be completed **within time limits**.

2. Requirements Elicitation

Process of **gathering information** from stakeholders to understand their needs.
Techniques used:

- Interviews
- Surveys and Questionnaires
- Focus Groups
- Observation
- Prototyping

Purpose: To gain complete domain knowledge and understand user expectations.

3. Requirements Specification

The process of **documenting all functional and non-functional requirements** in a clear and structured form (e.g., SRS Document).

Types of Requirements:

- **Functional:** Define *what* the system should do (e.g., data input, processing).
- **Non-functional:** Define *how well* the system performs (e.g., performance, reliability).
- **Constraints:** Define system limitations.
- **Acceptance Criteria:** Define conditions for successful completion.

Use of models: **DFDs, ER diagrams, FDDs**, etc.

4. Requirements Verification and Validation (V&V)

Ensures requirements are **correct, complete, and aligned** with stakeholder needs.

- **Verification:** Checks if requirements are consistent, clear, and error-free.
- **Validation:** Ensures that requirements fulfill stakeholder expectations.
- Methods: Reviews, walkthroughs, test cases, prototyping.
- Conducted iteratively throughout the SDLC.

5. Requirements Management

Involves **tracking, controlling, and updating requirements** throughout the project.

Key Activities:

- Change tracking and version control.
- Traceability between requirements and project components.
- Effective communication with stakeholders.
- Monitoring and reporting project progress.

Purpose: To handle **requirement changes** efficiently and avoid **scope creep**.

2) Requirements Elicictation

Definition

Requirements elicitation is the process of gathering and defining the needs and expectations of users for a software system. It ensures that the development process is based on a clear and complete understanding of customer requirements. It is one of the most communication-intensive, error-prone, and critical activities in software engineering.

Key Points

1. Successful elicitation requires an effective partnership between the customer and the developer.

2. It involves identification, collection, analysis, and refinement of requirements.
3. It is performed at the beginning of the Software Development Life Cycle (SDLC).
4. Various stakeholders are involved, such as users, business owners, developers, and domain experts.
5. It produces a clear, concise, and well-defined set of requirements for design and development.
6. Simple questioning is often insufficient, especially for safety-critical systems.
7. Techniques include interviews, surveys, observation, brainstorming, role-playing, and prototyping.

Importance of Requirements Elicitation

1. Ensures that software aligns with business objectives.
2. Improves user satisfaction by involving users early in the process.
3. Reduces rework, saving both time and cost.
4. Ensures compliance with legal and regulatory requirements.
5. Provides traceability and proper documentation for future reference.

Activities in Requirements Elicitation

1. Understanding the application domain – studying the environment in which the system will operate.
2. Understanding the customer's specific problem – identifying user needs and challenges.
3. Identifying external interactions – defining how the system interacts with other systems.
4. Conducting detailed user studies – observing and interviewing users to capture their requirements.
5. Defining constraints – identifying technical, legal, and operational limitations.

Methods of Requirements Elicitation

1. Interviews

Used to understand customer expectations and collect information directly from stakeholders.
There are two types of interviews:

- Open-ended interviews: No fixed agenda; allows free discussion.
- Structured interviews: Conducted with predetermined questions or a questionnaire.

2. Brainstorming Sessions

A group activity to generate a wide range of ideas in a short time.
A facilitator ensures that discussions remain focused and that all ideas are documented before prioritization.

3. Facilitated Application Specification Technique (FAST)

Aims to reduce the expectation gap between the customer and the developer.
Each participant lists objects related to the system, such as those in the environment, produced by the system, or used by the system.
The lists are combined, redundancies are removed, and a consolidated draft specification is prepared.

4. Quality Function Deployment (QFD)

Focuses on maximizing customer satisfaction.
It identifies three types of requirements:

- Normal requirements: Explicitly stated by users.
- Expected requirements: Implicit but essential features.
- Exciting requirements: Unexpected features that delight users.

5. Use Case Approach

Describes the functional behavior of a system from the user's perspective.
Key components include:

- Actor: An external entity interacting with the system.
  - Primary actor: Requires help from the system to achieve a goal.
  - Secondary actor: Provides a service to the system.
- Use Case: A description of interactions between actors and the system.
- Use Case Diagram: A visual representation showing actors, use cases, and their relationships.

Steps in Requirements Elicitation

1. Identify stakeholders – determine all participants such as users, customers, and developers.
2. Gather requirements – collect both functional and non-functional requirements using suitable techniques.
3. Prioritize requirements – classify them based on importance and urgency (for example, must-have, should-have, could-have, won't-have).
4. Categorize feasibility – determine whether requirements are achievable, deferred, or impossible based on project constraints.

Features of Requirements Elicitation

1. Involves all stakeholders for better understanding.
2. Gathers information about business processes and user environment.
3. Helps in prioritizing requirements based on value and feasibility.
4. Documents requirements in a clear and structured manner.
5. Includes validation and verification to ensure correctness.
6. Is iterative in nature and refined through feedback.

7. Promotes communication and collaboration.
8. Is flexible and adaptable to changing needs.

Advantages

1. Produces clear and refined requirements.
2. Improves communication between stakeholders.
3. Leads to development of high-quality software.
4. Prevents misunderstandings and reduces rework.
5. Helps in early identification of risks.
6. Aids in accurate project planning.
7. Builds user confidence in the software product.
8. May uncover new business opportunities.

Disadvantages

1. The process is time-consuming and costly.
2. Requires skilled and experienced analysts.
3. Requirements may change frequently.
4. Organizational politics or conflicts may influence outcomes.
5. Lack of stakeholder commitment can lead to poor results.
6. Conflicting priorities among stakeholders can cause delays.
7. Incomplete or inaccurate requirements may arise if not managed well.
8. Poor elicitation can lead to increased development costs and project delays.

3) Developing Use Cases

Before starting any project, it is essential to have a clear understanding of what needs to be done and how it will be accomplished. In software and systems engineering, a **use case** is a list of actions or event steps that define the interactions between a role (known as an *actor* in the Unified Modeling Language) and a system to achieve a specific goal.

An actor can be a human user, an external system, or even time. In systems engineering, use cases are used at a higher level than in software engineering, often representing missions or stakeholder goals.

In simple terms, a use case describes how a real-world actor interacts with a system. A **system use case** includes high-level implementation details and can be written either in an informal or formal manner.

Importance of Use Cases

Use cases have been widely used in software development for several decades. The main advantages and importance of use cases are as follows:

1. The list of goal names provides a concise summary of what the system will offer.
2. They give a clear overview of the roles of each component in the system, helping to define the roles of users, administrators, and other entities.

3. They help in clearly defining and understanding the user's needs and how the system should respond to them.
4. They provide structured answers to questions that might arise if a project begins without proper planning.

How to Plan a Use Case

The following example illustrates how to plan a use case.

1. **Use Case:** Defines the main objective of the use case. For example, adding a software component or adding a specific functionality.
2. **Primary Actor:** Identifies who will have access to the use case. For example, in the above cases, an administrator may be the primary actor.
3. **Scope:** Specifies the scope of the use case and its boundaries within the system.
4. **Level:** Describes the level of implementation for the use case (high-level or detailed).
5. **Flow:** Explains the functional flow or workflow of the use case, describing the sequence of interactions or steps involved.

Additional elements that can be included in a use case are:

- Preconditions
- Postconditions
- Brief course of action
- Time period

Use Case Diagram

A **use case diagram** is a visual representation of the interactions between actors and use cases in a system.

For example, consider a sample project similar to a social media platform like Facebook. The main actors in such a system could be the **User** and the **System**.

- During the **Sign Up** process, only the user interacts with the system.
- For actions such as **categorizing posts**, the system performs the operation without user intervention.

The diagram helps in understanding the relationship between various actors and the corresponding use cases within the system. It serves as a blueprint for developers to visualize the system's functional requirements.

4) Building the Analysis Model

An **Analysis Model** is a **technical representation of the system** that acts as a bridge between the **system description** (requirements) and the **design model** (architecture). It helps in defining and understanding the **information, behavior, and functions** of a system, which are later translated into architecture, component, and interface-level design during the design phase.

Objectives of Analysis Modelling

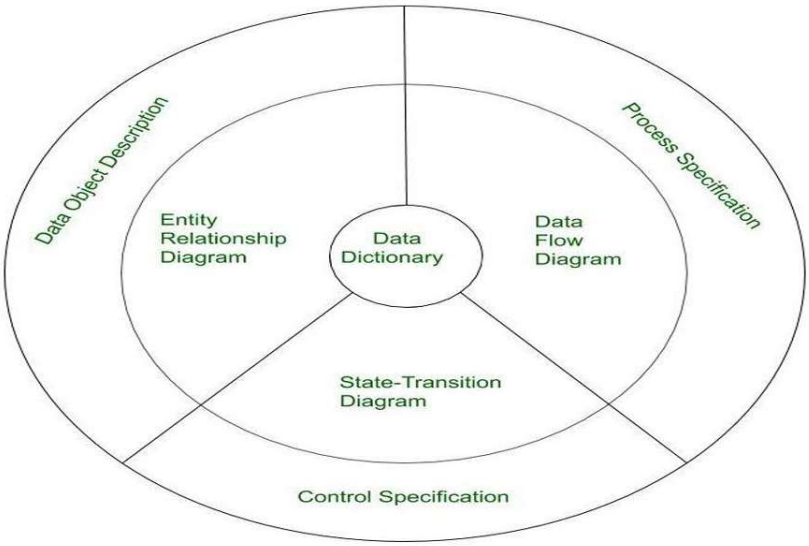
1. **Understanding Needs:**
   Helps in identifying and extracting user needs and requirements for the software system.
2. **Communication:**
   Serves as a medium of communication among users, clients, developers, and testers.
3. **Clarifying Ambiguities:**
   Assists in resolving misunderstandings and unclear requirements during the early stages.
4. **Finding Data Requirements:**
   Identifies relationships, entities, and attributes of the data required by the system.
5. **Defining Behavior:**
   Describes how the system behaves dynamically — including workflows, processes, and inter-component interactions.
6. **System Boundary Identification:**
   Defines the boundaries of the software system and how it interacts with users, other systems, or hardware components.

Elements of the Analysis Model

1. **Data Dictionary:**
   - A centralized repository containing descriptions of all data objects used or produced by the software.
   - Helps in modeling and managing data objects defined during the requirements stage.
2. **Entity Relationship Diagram (ERD):**
   - Depicts relationships between data objects.
   - Each object's attributes are described using the data object description.
   - Forms the basis for data design activities.
3. **Data Flow Diagram (DFD):**
   - Shows how data moves through the system and how it is transformed from input to output.
   - Provides information about the information domain and forms the foundation for modeling system functions.
   - Enables simultaneous modeling of functional and informational aspects of the system.
4. **State Transition Diagram:**
   - Represents different states of a system and transitions between those states based on events.

- Shows how the system behaves in response to external inputs.
- Describes the actions taken when specific events occur.

5. **Process Specification (PSPEC):**
   - Describes each process or function represented in the DFD.
   - Specifies inputs, algorithms or transformations, and resulting outputs.
   - Includes performance constraints and layout limitations affecting process execution.
6. **Control Specification (CSPEC):**
   - Represents the control aspects of the software.
   - Describes how the system reacts to events and which processes are triggered by them.
   - Helps manage event-driven behavior and process execution.
7. **Data Object Description:**
   - Provides detailed information about each data object used in the software.
   - Includes details about attributes of data objects defined in the ERD.
   - Serves as a comprehensive reference for all data entities within the system.



Key Principles of Analysis Modelling

1. **Abstraction:**
   Focuses on essential system concepts, behavior, and relationships while ignoring unnecessary details.
2. **Modularity:**
   Divides the system into smaller, manageable modules or components that represent different functionalities.
   This makes understanding, evaluation, and modification easier.
3. **Consistency:**
   Ensures that the analysis model is consistent internally and aligns with other artifacts like requirement documents, design specifications, and code.
   Consistency avoids conflicts and improves stakeholder understanding.
4. **Traceability:**
   Provides the ability to trace requirements from their origin through implementation.

This supports change management, impact analysis, and validation throughout the project lifecycle.

5. **Precision:**
   Ensures that the model accurately and clearly represents system requirements and behaviors.
   Precision helps prevent miscommunication and implementation errors.
6. **Separation of Concerns:**
   Divides the system into distinct areas of focus.
   For example, data modeling concentrates on structure and relationships of data, while behavioral modeling captures dynamic system actions and interactions.

5) System Requirement Definition

What are Software Requirements?

According to **IEEE Standard 729**, a *requirement* is defined as:

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system or component to satisfy a contract, standard, or specification.
3. A documented representation of a condition or capability as stated above.

Types of Software Requirements

Software requirements are mainly classified into three types:

- **Functional Requirements**
- **Non-Functional Requirements**
- **Domain Requirements**

1. Functional Requirements

Definition:
Describe *what* the software should do — the specific features or functions the system must have.

Examples:

- User authentication (login with username and password)
- Search functionality (search by name or category)
- Report generation (sales report for a date range)

Explanation:
Functional requirements specify the actions and operations the system must perform.
They are visible to users and form the core behavior of the software.

Key Points:

- Represented as input, operation, and expected output.
- Described using natural or structured language.
- Also called *Functional Specifications*.

2. Non-Functional Requirements

Definition:
Describe *how* the system performs its functions — the quality attributes or constraints.

Examples:

- **Performance:** Process 1,000 transactions per second.
- **Usability:** Simple and user-friendly interface.
- **Reliability:** 99.9% system uptime.
- **Security:** Encrypt data during transmission and storage.

Explanation:
These define the performance, security, maintainability, and scalability of the system.

Common Non-Functional Aspects:

- Portability
- Security
- Maintainability
- Reliability
- Scalability
- Performance
- Reusability
- Flexibility

Categories:

- **Execution Qualities:** Observable at runtime (e.g., security, usability).
- **Evolution Qualities:** Related to system design (e.g., maintainability, scalability).

3. Domain Requirements

Definition:
Domain requirements are specific to a particular **industry or application domain**.
They capture terminology, standards, and rules unique to that domain.

Examples:

- **Healthcare:** Must comply with HIPAA for patient data.
- **Finance:** Follow GAAP for financial reporting.
- **E-commerce:** Support PayPal, Stripe, and credit card payments.

Explanation:
Domain requirements ensure that the software adheres to industry-specific needs and regulations.

Other Classifications of Software Requirements

1. **User Requirements:**
   - Describe what end-users expect from the system.
   - Usually written in natural language.
2. **System Requirements:**
   - Define technical specifications (hardware, software, architecture, interfaces).
3. **Business Requirements:**
   - Describe business goals (e.g., revenue growth, customer satisfaction).
4. **Regulatory Requirements:**
   - Define legal or compliance obligations (e.g., data privacy, accessibility).
5. **Interface Requirements:**
   - Specify interactions with external systems or databases.
6. **Design Requirements:**
   - Focus on architecture, algorithms, and data structures.

Advantages of Classifying Software Requirements

1. **Better Organization:** Easier to manage and track requirements.
2. **Improved Communication:** Clear understanding among stakeholders.
3. **Increased Quality:** Early identification of gaps or conflicts.
4. **Improved Traceability:** Easier to verify compliance and progress.

Disadvantages of Classifying Software Requirements

1. **Complexity:** Difficult when many stakeholders are involved.
2. **Rigid Structure:** May limit flexibility for changes.
3. **Misclassification:** Can lead to costly misunderstandings later.

6) System Architectural Design

Definition

According to **IEEE**, architectural design is:

"The process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system."

Software architecture represents the **blueprint** of the system, showing how components interact and integrate to perform desired functions.

System Category Consists of

1. **Components:**
   Examples – databases, computational modules, and other functional units.
   These perform the required operations of the system.
2. **Connectors:**
   Enable coordination, communication, and cooperation between components.
3. **Integration Conditions:**
   Define how components can be combined to form the complete system.
4. **Semantic Models:**
   Help designers understand overall properties and behaviors of the system.

Purpose of Architectural Styles

Architectural styles define **structural patterns** for software systems.
They provide a common framework that simplifies understanding, design, and modification.

Taxonomy of Architectural Styles

1. Data-Centered Architecture

- A **central data store (repository)** is shared by various components that access and modify it.
- Client software interacts with the central repository.
- The "blackboard" model notifies clients when data changes.

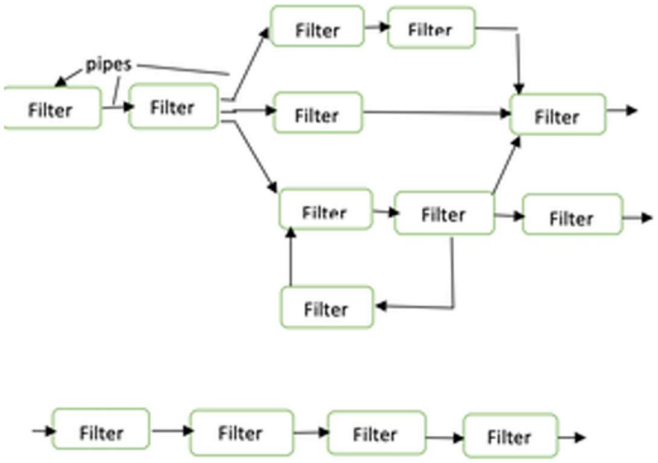
Advantages:

- Data repository is independent of clients.
- Clients work independently.
- Easy to add or modify clients.
- Simplifies maintenance and integration.

2. Data-Flow Architecture (Pipe and Filter)

- Used when **input data is transformed into output data** through a series of processing steps.

- Consists of **filters (processing components)** connected by **pipes (data transmission lines)**.
- Each filter works independently, taking input and producing output for the next filter.



Advantages:

- Supports concurrent execution.
- Encourages easy modification and reuse.

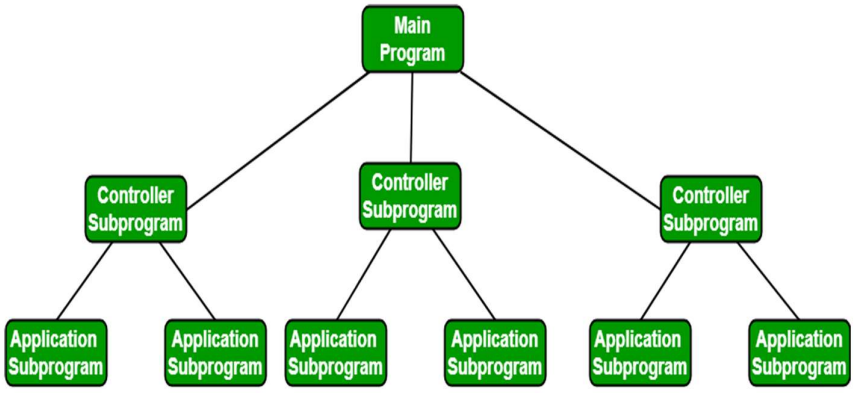
Disadvantages:

- May degrade into a batch sequential system.
- Not suitable for applications needing high user interaction.
- Difficult to coordinate multiple data streams.

3. Call and Return Architecture

Used to design **modular and scalable programs**.
It consists of sub-styles such as:

- **Main Program–Subprogram Architecture:**
  The main program controls several subprograms or modules in a hierarchical structure.
- **Remote Procedure Call (RPC) Architecture:**
  Components are distributed across multiple computers; functions are invoked remotely.



Advantages:

- Simplifies debugging and maintenance.
- Enhances modularity and scalability.

4. Object-Oriented Architecture

- The system is built using **objects** that encapsulate both data and operations.
- Objects interact through **message passing**.

Characteristics:

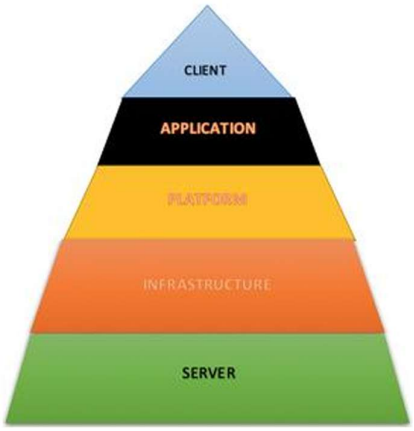
- Objects protect system integrity.
- Each object is unaware of internal representations of others.

Advantages:

- Simplifies problem-solving by dividing into independent objects.
- Supports modification without affecting other components.
- Promotes reusability and flexibility.

5. Layered Architecture

- The system is structured into **multiple layers**, each performing a defined set of operations.
- Outer layers handle **user interfaces**, while inner layers manage **system-level operations**.



Structure:

- **Outer Layer:** User interface handling.
- **Intermediate Layers:** Utility services and application functions.
- **Inner Layer:** OS interfacing and system communication.

Example:
OSI-ISO model (Open Systems Interconnection – International Organization for Standardization).

Advantages:

- Simplifies debugging and maintenance.
- Enhances separation of concerns.
- Supports portability and scalability.

Summary

Architecture Style	Key Idea	Main Advantage
Data-Centered	Central data repository shared by clients	Easy to add/modify clients
Data-Flow	Sequential data transformation via filters	Supports concurrency
Call & Return	Hierarchical modular structure	Easy to modify and scale
Object-Oriented	Interaction via message-passing between objects	Promotes reuse and flexibility
Layered	Hierarchical layers with defined responsibilities	Enhances maintainability

7) Subsystems Development

Subsystem development involves dividing a large system into smaller, manageable units or **subsystems**, each handling a specific function.
Each subsystem is developed, tested, and verified independently before being integrated into the main system.
This modular approach improves maintainability, parallel development, and simplifies debugging.

System Integration Testing (SIT)

Definition

System Integration Testing (SIT) is a testing process conducted to verify the interaction and integration of multiple software and/or hardware components as a complete system.
It ensures that all integrated modules function together correctly.

SIT is performed **after unit and integration testing** and **before User Acceptance Testing (UAT)**.

Key Characteristics

- Black-box testing technique.
- Verifies data flow, control flow, and interactions among modules.
- Focuses on both **functional** and **non-functional** requirements.
- Test environment closely resembles the **production environment**.

Objectives of SIT

- Ensure the system meets user requirements.
- Verify end-to-end data and control flow.
- Identify errors, defects, and incompatibilities.
- Optimize system memory and performance.
- Reduce overall testing time and rework.

Major States of SIT

1. **Data State within the Integration Layer**
    - Handles data transmission between components.
    - Uses middleware and web services for transformation and validation.
2. **Data State within the Database Layer**
    - Validates data transferred from integration layer to database.
    - Ensures correct data properties and SQL-based storage operations.
3. **Data State within the Application Layer**
    - Maps database data to the user interface.
    - Checks data consistency and interaction with front-end components.

Advantages of SIT

1. Ensures full system functionality and requirement compliance.
2. Detects and resolves integration issues early.
3. Improves overall system quality and performance.
4. Enhances collaboration between development and testing teams.
5. Reduces project risk and costly rework.
6. Supports Agile and continuous development approaches.

System Configuration Management (SCM)

Definition

System Configuration Management (SCM) is the discipline of **controlling, recording, and tracking** all changes made to a software system during its lifecycle.
It ensures that every modification is authorized, documented, and implemented systematically.

SCM is essential for maintaining the **stability, integrity, and traceability** of evolving software systems.

Key Processes in SCM

1. **Identification and Establishment**
   - Identify configuration items (code, documents, components).
   - Define relationships and control mechanisms.
   - Create baselines as reference points for future changes.
2. **Version Control**
   - Manage multiple versions and evolution paths of software items.
   - Track minor and major changes systematically (e.g., version 1.0 → 1.1 → 2.0).
3. **Change Control**
   - Evaluate change requests (CR) for technical feasibility, impact, and cost.
   - Approved changes become Engineering Change Requests (ECR).
   - Changes are implemented, tested, and checked into the system with version updates.
4. **Configuration Auditing**
   - Verifies technical correctness and completeness of configuration items.
   - Ensures consistency and compliance with defined baselines.
5. **Reporting**
   - Provides status and version updates to all stakeholders through documentation such as release notes, guides, and reports.

Importance of SCM

- Enables effective bug tracking and issue resolution.
- Supports continuous integration and deployment.
- Reduces risk of defects in production.
- Facilitates parallel development by multiple teams.
- Maintains reproducibility and traceability across versions.
- Essential for managing large and complex software projects.

Need for SCM

1. **Replicability:** Enables reproducing any version for testing or debugging.
2. **Identification:** Tracks and labels all configuration items and their relationships.
3. **Efficiency:** Automates versioning, merging, and dependency management to improve productivity.

Objectives of SCM

1. Control the evolution and modification of software systems.
2. Enable collaboration and synchronization among teams.
3. Provide complete version control and rollback capability.
4. Facilitate reliable replication and distribution of software.
5. Improve quality, reliability, and consistency of software systems.