UNIT I

Software Engineering Concepts: -

Software engineering is a discipline that applies engineering principles to the design, development, testing, and maintenance of software systems. It aims to produce high-quality software that is reliable, maintainable, and meets user needs.

Here are some key concepts in software engineering:

1. Software Development Life Cycle (SDLC)

Definition: A framework that outlines the stages involved in software development from conception to deployment and maintenance.

Phases involved in software development:

a) Requirement Analysis – Gathering and defining what the software needs to accomplish.

b) Design – Creating a blueprint for the software system's architecture and components.

c) Implementation (Coding) – Writing the actual code based on the design specifications.

d) Testing – Verifying that the software works as intended and identifying defects.

e) Deployment – Releasing the software to users and setting up the production environment.

f) Maintenance – Updating and fixing the software post-deployment.

2. Software Development Models

The phases of SDLC remain the same in all software development processes, but software engineers may choose to implement functions in every phase in different ways through different models.

The most common SDLC models include Waterfall, Agile, Scrum, Kanban etc.

Waterfall Model: A linear and sequential approach where each phase must be completed before moving to the next. It's simple but can be inflexible to changes.

Agile Methodology: An iterative and incremental approach that emphasizes flexibility, collaboration, and customer feedback. Common frameworks include Scrum and Kanban.

3. Requirements Engineering

Definition: The process of defining and managing software requirements. It involves identifying user needs and specifying what the software should do.

Techniques:

Interviews: Directly engaging with stakeholders to gather requirements.

Surveys and Questionnaires: Collecting requirements from a larger group.

Use Cases: Describing how users will interact with the system.

User Stories: Short, simple descriptions of features from an end-user perspective.

4. Software Design

Definition: The process of defining the architecture, components, interfaces, and other characteristics of a system.

5. Coding and Implementation

Definition: The actual process of writing and compiling code to build the software.

6. Testing

Definition: The process of evaluating software to ensure it meets requirements and is free of defects.

7. Software Maintenance

Definition: Activities required to keep the software operational after deployment, including fixing bugs, updating features, and optimizing performance.

9. Software Metrics

Definition: Quantitative measures used to assess various aspects of software development and performance.

Software Engineering Principles: -

The following set of core principles is fundamental to the practice of software engineering:

Principle 1. Divide and conquer. Stated in a more technical manner, analysis and design should always emphasize separation of concerns (SoCs). A large problem is easier to solve if it is subdivided into a collection of elements (or concerns).

Principle 2. Understand the use of abstraction. At its core, an abstraction is a simplification of some complex element of a system used to communicate meaning in a single phrase. In software engineering practice, you use many different levels of abstraction, each imparting or implying meaning that must be communicated. In analysis and design work, a software team normally begins with models that represent high levels of abstraction (e.g., a spreadsheet) and slowly refines those models into lower levels of abstraction (e.g., a column or the SUM function).

Principle 3. Strive (make great effort) for consistency. Whether it's creating an analysis model, developing a software design, generating source code, or creating test cases, the principle of

consistency suggests that a familiar context makes software easier to use. For example, consider the design of a user interface for a mobile app. Consistent placement of menu options, the use of a consistent color scheme, and the consistent use of recognizable icons help to create a highly effective user experience.

Principle 4. Focus on the transfer of information. Software is about information transfer from a database to an end user, from an end user into a graphic user interface (GUI), from an operating system to an application, from one software component to another etc. In every case, information flows across an interface, and this means there are chances for errors, omissions, or ambiguity. The implication of this principle is that you must pay attention to the analysis, design, construction, and testing of interfaces.

Principle 5. Build software that exhibits effective modularity. That is, each module should focus exclusively on one well-constrained aspect of the system. Additionally, modules should be interconnected in a relatively simple manner to other modules, to data sources, and to other environmental aspects.

Principle 6. Look for patterns. Software engineers use patterns as a means of cataloging and reusing solutions to problems they have encountered in the past. The use of these design patterns can be applied to wider systems engineering and systems integration problems, by allowing components in complex systems to evolve independently.

Principle 7. When possible, represent thae problem and its solution from several different perspectives. When a problem and its solution are examined from different perspectives, it is more likely that greater insight will be achieved and that errors and omissions will be uncovered.

Software Development Life Cycle (SDLC): -

The Software Development Life Cycle (SDLC) is a structured process that software developers and engineers use to design, develop, test, and deploy software applications. It provides a systematic approach to building software, ensuring quality and efficiency at each stage. The SDLC typically consists of several distinct phases, each with specific objectives and deliverables.

Here's an overview of the main phases in the SDLC:

1. Planning and Requirement Analysis

Objective: Define the project's scope, objectives, and feasibility. Gather and analyze the requirements.

Activities:

Stakeholder meetings to gather requirements and expectations.

Feasibility analysis to assess the technical, economic, and operational viability.

Requirement specification to document the functional and non-functional requirements.

Deliverables:

Project plan

Requirement specification document

Feasibility study report

2. System Design

Objective: Design the architecture and detailed specifications of the system based on the requirements.

Activities:

High-level design (HLD) to outline the system architecture, modules, and data flow.

Low-level design (LLD) to specify the details of individual components, including data structures, algorithms, and interfaces.

Design of user interfaces, databases, and system security.

Deliverables:

High-level design document

Low-level design document

Database design

User interface design

3. Implementation (Coding)

Objective: Convert the design specifications into working code.

Activities:

Writing code based on the design documents.

Adhering to coding standards and guidelines.

Using version control systems to manage code changes.

Deliverables:

Source code

Code documentation

Code repositories

4. Testing

Objective: Verify that the software meets the specified requirements and identify any defects.

Activities:

Unit testing to verify the functionality of individual components.

Integration testing to check the interactions between integrated modules.

System testing to validate the complete system against requirements.

User Acceptance Testing (UAT) to ensure the system meets user needs and requirements.

Deliverables:

Test plans and cases

Test scripts

Test reports

5. Deployment

Objective: Deploy the software to the production environment and make it available to users.

Activities:

Preparing the production environment, including hardware, network, and software configurations.

Installing and configuring the software.

Performing final system checks and validation.

Training users and providing documentation.

Deliverables:

Deployed software

Deployment guides

User manuals and training materials

6. Maintenance and Support

Objective: Ensure the software continues to function correctly after deployment and meets evolving user needs.

Activities:

Monitoring the system for any issues or bugs.

Providing regular updates, patches, and enhancements.

Handling user support requests and feedback.

Performing preventive maintenance to prevent future issues.

Deliverables:

Maintenance logs

Update and patch documentation

User support records

SDLC Models:

There are several SDLC models, each suited to different types of projects and organizational needs. Some of the most common models include:

Waterfall Model: A linear and sequential approach, where each phase must be completed before the next begins. It is simple but less flexible to changes.

Agile Model: An iterative and incremental approach that emphasizes flexibility, collaboration, and customer feedback. It includes frameworks like Scrum and Kanban.

Agile software development model: -

The Agile software development model is a popular and flexible approach to software system development that emphasizes iterative progress, collaboration, and adaptability.

Here are the key principles and practices that define the Agile methodology:

Key Principles:

Individuals and Interactions over Processes and Tools: Agile values the skills, creativity, and communication of team members more than the tools they use.

Working Software over Comprehensive Documentation: The primary measure of progress is working software. Documentation is important, but delivering software that works is more critical.

Customer Collaboration over Contract Negotiation: Agile emphasizes regular collaboration with customers to understand their needs and adapt the project accordingly.

Responding to Change over Following a Plan: Agile methodologies are designed to be flexible and adaptable, responding to changes even late in development.

Agile Practices:

Iterative Development: Projects are broken down into small iterations or sprints, typically lasting 2-4 weeks. Each iteration involves planning, designing, coding, testing, and reviewing a subset of the final product.

Continuous Feedback: Regular feedback from stakeholders and end-users is incorporated into the development process to ensure the product meets their needs.

Daily Standups: Short, daily meetings where team members discuss what they did the previous day, what they plan to do today, and any blockers they are facing.

User Stories: Requirements are captured as user stories, which describe the desired functionality from the end-user perspective. User stories are prioritized and managed in a product backlog.

Sprint Planning and Review: Each sprint begins with a planning meeting where the team selects the user stories to work on. At the end of the sprint, a review meeting is held to demonstrate the completed work to stakeholders.

Retrospectives: After each sprint, the team holds a retrospective meeting to discuss what went well, what didn't, and how processes can be improved in the next sprint.

Cross-Functional Teams: Agile teams are typically composed of members with different skills (e.g., developers, testers, designers) to foster collaboration and ensure all aspects of the product are addressed.

Popular Agile Frameworks –

Scrum: A framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value. Scrum includes roles (Scrum Master, Product Owner, Development Team), events (Sprint, Daily Scrum, Sprint Review, Sprint Retrospective), and artifacts (Product Backlog, Sprint Backlog, Increment).

Kanban: Focuses on visualizing the flow of work, limiting work in progress, and maximizing efficiency. It uses a Kanban board to visualize tasks and their status.

Scrum framework for software development: -

The Scrum framework is one of the most widely adopted Agile methodologies for managing and completing complex projects. It provides a structured approach to iterative development through defined roles, events, and artifacts.

Key Roles:

Product Owner:

Represents the stakeholders and the voice of the customer.

Responsible for defining and prioritizing the product backlog.

Ensures the team is working on the most valuable features.

Scrum Master:

Facilitates the Scrum process and ensures that the team adheres to Scrum practices.

Helps resolve impediments that the team encounters.

Acts as a coach for the team, ensuring that Agile principles are followed.

Development Team:

A cross-functional group of professionals (developers, testers, designers) responsible for delivering potentially shippable increments of the product. Self-organizing and responsible for managing their work.

Key Events:

Sprint: A time-boxed iteration, typically 2-4 weeks, during which a potentially shippable product increment is created. Consists of planning, execution, review, and retrospective phases.

Sprint Planning: A meeting held at the beginning of each sprint to determine what can be delivered in the upcoming sprint and how that work will be achieved.

The Product Owner presents the top items from the product backlog, and the team selects the items they can commit to.

Daily Scrum (Daily Standup): A short, daily meeting (usually 15 minutes) where the development team synchronizes their work and plans for the next 24 hours.

Team members answer three questions: What did I do yesterday? What will I do today? Are there any impediments in my way?

Sprint Review: Held at the end of the sprint to inspect the increment and adapt the product backlog if needed. The development team demonstrates what has been completed during the sprint. Stakeholders provide feedback.

Sprint Retrospective: A meeting held after the sprint review and before the next sprint planning to reflect on the sprint. The team discusses what went well, what didn't, and how processes can be improved.

Key Artifacts:

Product Backlog: An ordered list of everything that is known to be needed in the product. Managed by the Product Owner and continuously refined.

Sprint Backlog: A list of items selected from the product backlog to be completed in the current sprint, along with a plan for delivering the product increment and achieving the sprint goal.

Increment: The sum of all the product backlog items completed during a sprint and all previous sprints. Must be in a usable condition regardless of whether the Product Owner decides to release it.

Backlog Refinement: The Product Owner and development team regularly refine the product backlog to ensure that it is well-defined and prioritized.

Sprint Planning: At the start of each sprint, the team selects items from the product backlog to move to the sprint backlog and defines a sprint goal.

Daily Scrum: The team meets daily to discuss progress, plan the day's work, and identify any impediments.

Sprint Execution: The development team works on the tasks in the sprint backlog to create a potentially shippable product increment.

Sprint Review: At the end of the sprint, the team presents the increment to stakeholders for feedback.

Sprint Retrospective: The team reflects on the sprint and identifies improvements for the next sprint.

Kanban framework for software development: -

The Kanban framework is a visual process management system that helps teams visualize their work, limit work in progress, and maximize efficiency or flow. Unlike Scrum, which has predefined roles and events, Kanban is more flexible and can be applied to various existing workflows without necessitating drastic changes.

Key Principles:

Visualize Work: Use a Kanban board to represent work items and their status. This visualization helps the team understand the flow of work and identify bottlenecks.

Limit Work in Progress (WIP): Restrict the number of work items in progress at any stage of the workflow to ensure focus and efficiency.

Manage Flow: Continuously monitor and manage the flow of work items from start to finish to optimize throughput.

Make Process Policies Explicit: Clearly define and communicate the process policies so that everyone understands how work should be done.

Implement Feedback Loops: Regularly review and adapt processes based on feedback to improve continuously.

Improve Collaboratively, Evolve Experimentally: Use data and collaborative discussion to drive continuous improvement in small, incremental changes.

Key Components:

Kanban Board: The central tool in Kanban, which visually represents the workflow. It typically includes columns such as "To Do," "In Progress," and "Done," but can be customized to fit the team's process.

Cards: Represent individual work items or tasks. Each card includes information about the task, such as a description, assignee, due date, and other relevant details.

WIP Limits: Set maximum limits for the number of work items in each column (e.g., no more than 3 items in "In Progress") to prevent overloading and to focus on finishing tasks.

Visualize Workflow: Map out the entire workflow on the Kanban board, from initial request to delivery. Break down the workflow into distinct stages or columns.

Create and Move Cards: Each work item is represented by a card that moves across the board from left to right as it progresses through different stages.

Set WIP Limits: Establish WIP limits for each column to ensure that the team does not take on too much work at once, maintaining focus and productivity.

Monitor and Manage Flow: Regularly review the board to identify bottlenecks, track progress, and manage the flow of work. Ensure that work items are moving smoothly from one stage to the next.

Continuous Improvement: Use metrics such as lead time (time from task initiation to completion) and cycle time (time from start of work to completion) to identify areas for improvement. Hold regular meetings to discuss process improvements.

Requirements Engineering: -

Requirement engineering begins with inception (a task that defines the scope and nature of the problem to be solved). It moves onward to elicitation (a task that helps stakeholders define what is required), and then elaboration (where basic requirements are refined and modified). As stakeholders define the problem, negotiation occurs (what are the priorities, what is essential, when is it required?). Finally, the problem is specified in some manner and then reviewed or validated to ensure that your understanding of the problem and the stakeholders' understanding of the problem coincide.

Requirement engineering

Requirement engineering is the term for the broad spectrum of tasks and techniques that lead to an understanding of requirements.

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. Requirements Engineering establishes a base for design and construction.

Without it, the resulting software has a high probability of not meeting customer's needs. It must be adapted to the needs of the process, the project, the product. It is important to realize that each of these tasks is done iteratively as the project team and the stakeholders(e.g., managers, customers, and end users) continue to share information about their respective concerns.

Inception:

How does a software project get started? In general, most projects begin with an identified business need or when a potential new market or service is discovered.

At project inception, you establish a basic understanding of the problem, the people who want a solution, and the nature of the solution that is desired. Communication between all stakeholders and the software team needs to be established during this task to begin an effective collaboration.

Elicitation:

Here, asks the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis. But it isn't simple—it's very hard.

An important part of elicitation is to understand the business goals. A goal is a long-term aim that a system or product must achieve. Goals may deal with either functional or nonfunctional (e.g., reliability, security, usability) concerns.

Goals are often a good way to explain requirements to stakeholders and, once established, can be used to manage conflicts among stakeholders.

Goals should be specified precisely and serve as the basis for requirements elaboration, verification and validation, conflict management, negotiation, explanation, and evolution.

Once the goals are captured, you establish a prioritization mechanism and create a design rationale for a potential architecture (that meets stakeholder goals).

Agility is an important aspect of requirements engineering. The intent of elicitation is to transfer ideas from stakeholders to the software team smoothly and without delay.

It is highly likely that new requirements will continue to emerge as iterative product development occurs.

Elaboration:

The elaboration task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information. Elaboration is driven by the creation and refinement of user scenarios obtained during elicitation.

These scenarios describe how the end users (and other actors) will interact with the system. Each user scenario is parsed to extract analysis classes—business domain entities that are visible to the end user. The attributes of each analysis class are defined, and the services that are required by each class are identified.

The relationships and collaboration between classes are identified. Elaboration is a good thing, but you need to know when to stop. The key is to describe the problem in a way that establishes a firm base for design and then move on. Do not obsess over unnecessary details.

Specification:

In the context of computer-based systems (and software), the term specification means different things to different people.

A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

Some suggest that a "standard template" should be developed and used for a specification, arguing that this leads to requirements that are presented in a consistent and therefore more understandable manner.

However, it is sometimes necessary to remain flexible when a specification is to be developed. The formality and format of a specification varies with the size and the complexity of the software to be built.

For large systems, a written document, combining natural language descriptions and graphical models, may be the best approach.

Validation:

The work products produced during requirements engineering are assessed for quality during a validation step. A key concern during requirements validation is consistency.

Use the analysis model to ensure that requirements have been consistently stated. Requirement's validation examines the specification to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

The primary requirements validation mechanism is the technical review. The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content or interpretation, areas where

clarification may be required, missing information, inconsistencies (a major problem when large products or systems are engineered), conflicting requirements, or unrealistic (unachievable) requirements.

Ethics and Professional Practices: -

Software engineering has evolved into a respected, worldwide profession. As professionals, software engineers should abide by a code of ethics that guides the work they do and the products they produce. An ACM/IEEE-CS Joint Task Force has produced a Software Engineering Code of Ethics and Professional Practices (Version 5.1).

The code [ACM12] states:

1. PUBLIC—Software engineers shall act consistently with the public interest.

2. CLIENT AND EMPLOYER—Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.

3. PRODUCT—Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.

4. JUDGMENT—Software engineers shall maintain integrity and independence in their professional judgment.

5. MANAGEMENT—Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.

6. PROFESSION—Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.

7. COLLEAGUES—Software engineers shall be fair to and supportive of their colleagues.

8. SELF—Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Software Requirements Specification (SRS): -

A software requirements specification (SRS) is a description of a software system to be developed. It lays out functional and non-functional requirements and may include a set of use cases that describe user interactions that the software must provide.

Introduction

Purpose of this Document – At first, main aim of why this document is necessary and what's purpose of document is explained and described.

Scope of this document – In this, overall working and main objective of document and what value it will provide to customer is described and explained. It also includes a description of development cost and time required.

Overview – In this, description of product is explained. It's simply summary or overall review of product.

General description

In this, general functions of product which includes objective of user, a user characteristic, features, benefits, its importance. It also describes features of user community.

Functional Requirements

In this, possible outcome of software system which includes effects due to operation of program is fully explained. All functional requirements which may include calculations, data processing, etc. are placed in a ranked order. Functional requirements specify the expected behavior of the system-which outputs should be produced from the given inputs.

They describe the relationship between the input and output of the system. For each functional requirement, detailed description all the data inputs and their source, the units of measure, and the range of valid inputs must be specified.

Interface Requirements

In this, software interfaces which mean how software program communicates with each other or users either in form of any language, code, or message are fully described and explained. Examples can be shared memory, data streams, etc.

Performance Requirements

In this, how a software system performs desired functions under specific condition is explained. It also explains required time, required memory, maximum error rate, etc. All the requirements relating to the performance characteristics of the system must be clearly specified. There are two types of performance requirements: static and dynamic.

Design Constraints

In this, constraints limitation or restriction are specified and explained for design team. Examples may include use of a particular algorithm, hardware and software limitations, etc. There are a number of factors in the client's environment that may restrict the choices of a designer leading to design constraints such factors include standards that must be followed resource limits, operating environment, reliability and security requirements and policies that may have an impact on the design of the system.

Non-Functional Attributes

In this, non-functional attributes are explained that are required by software system for better performance. An example may include Security, Portability, Reliability, Reusability, Application compatibility, Data integrity, Scalability capacity, etc.

Preliminary Schedule and Budget

In this, initial version and budget of project plan are explained which include overall time duration required and overall cost required for development of project.

Appendices

In this, additional information like references from where information is gathered, definitions of some specific terms, acronyms, abbreviations, etc. are given and explained.

Uses of SRS document

1. Development team require it for developing product according to the need.

2. Test plans are generated by testing group based on the describe external behavior.

3. Maintenance and support staff need it to understand what the software product is supposed to do.

4. Project manager base their plans and estimates of schedule, effort and resources on it.

5. customer rely on it to know that product they can expect.

6. As a contract between developer and customer.

SRS's characteristics include:

Correct

Unambiguous

Complete

Consistent

Ranked for importance and/or stability

Verifiable

Modifiable

Traceable