SNS COLLEGE OF TECHNOLOGY

Coimbatore-35.

An Autonomous Institution

COURSE NAME : 19CST201 AGILE SOFTWARE ENGINEERING

II YEAR/ III SEMESTER

UNIT – I INTRODUCTION TO SOFTWARE ENGINEERING

UNIT I INTRODUCTION TO SOFTWARE ENGINEERING

The Nature of Software -Software Engineering - Software engineering Practice – Process Models: Generic – Prescriptive – Specialized - United Process - Personal and Team Process Models - Process Technology-Understanding Requirements-Design concepts & model-Software quality concepts & Review metrics.

Process Technology

To implement software process models, process technology tools are developed to help software organizations :

- analyze their current process
- organize work tasks
- control and monitor progress
- manage technical quality.
- to build an automated model of the process framework, task sets.

Process Technology

- Once an acceptable process has been created, other process technology tools can be used to allocate, monitor, and even control all software engineering activities, actions, and tasks defined as part of the process model.

- Each member of a software team can use such tools to develop a checklist of work tasks to be performed, work products to be produced, and quality assurance activities to be conducted.

- The process technology tool can also be used to coordinate the use of other software engineering tools that are appropriate for a particular work task.

Understanding Requirements

1. Requirements Engineering
2. Establishing the Groundwork
3. Eliciting Requirements
4. Developing Use Cases
5. Building the Requirements Model
6. Negotiating Requirements
7. Validating Requirements

Understanding Requirements

1. Requirements Engineering

The broad spectrum of tasks and techniques that lead to an understanding of requirements is called requirements engineering.

- **Inception**

- **Elicitation**
  - Problems of scope
  - Problems of understanding
  - Problems of volatility

- **Elaboration**
- **Negotiation**
- **Specification**
- **Validation**

Understanding Requirements

2. Establishing the Groundwork

- Identifying Stakeholders
- Recognizing Multiple Viewpoints
- Working toward Collaboration
- Asking the First Questions

3. Eliciting Requirements

- Collaborative Requirements Gathering
- Quality Function Deployment
- Usage Scenarios
- Elicitation Work Products

Understanding Requirements

4. Developing Use Cases :

❖ A use case diagram is used to represent the dynamic behavior of a system. It encapsulates the system's functionality by incorporating use cases, actors, and their relationships.

❖ Purposes of a use case diagram :

- It gathers the system's needs.
- It depicts the external view of the system.
- It recognizes the internal as well as external factors that influence the system.
- It represents the interaction between the actors.

Understanding Requirements

Questions that should be answered by a use case:

- Who is the primary actor, the secondary actor(s)?
- What are the actor's goals?
- What preconditions should exist before the story begins?
- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the story is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

Understanding Requirements

5. Building the Requirements Model :

❖ Elements of the Requirements Model

- **Scenario-based elements** - The system is described from the user's point of view using a scenario-based approach.

- **Class-based elements** - Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of things that have similar attributes and common behaviors.

Understanding Requirements

5. Building the Requirements Model :

❖ Elements of the Requirements Model

- **Behavioral elements -** design that is chosen and the implementation approach that is applied. State diagram is used.

- **Flow-oriented elements -** Information is transformed. The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms.

Understanding Requirements

❖ **Analysis Patterns :**

- Anyone who has done requirements engineering on more than a few software projects begins to notice that certain problems reoccur across all projects within a specific application domain.
- These *analysis patterns* suggest solutions (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications.

Benefits that can be associated with the use of analysis patterns:

- analysis patterns speed up the development of abstract analysis models by providing reusable analysis models with examples as well as a description of advantages and limitations.
- facilitate the transformation of the analysis model into a design model

Understanding Requirements

6. Negotiating Requirements

- Negotiation with one or more stakeholders - cost and time-to-market.
- Negotiation is to develop a project plan that meets stakeholder needs while at the same time reflecting the real-world constraints (e.g., time, people, budget) that have been placed on the software team.
- Best negotiations “win-win” result - stakeholders win by getting the system or product that satisfies the majority of their needs and members of the software team win by working to realistic and achievable budgets and deadlines.

7. Validating Requirements

- Examined for inconsistency, omissions, and ambiguity.

Design Concepts

- Abstraction
- Architecture
- Patterns
- Separation of Concerns
- Modularity
- Information Hiding

Design Concepts

- Functional Independence
- Refinement
- Aspects
- Refactoring
- Object-Oriented Design Concepts
- Design Classes

Design Concepts

➢ Abstraction

- Solution to any problem can have many levels of abstraction can be used.
- At the **highest level of abstraction**, a solution is stated in broad terms using the language of the problem environment.
- At **lower levels of abstraction**, a more detailed description of the solution is provided. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution.
- Finally, at the **lowest level of abstraction**, the solution is stated in a manner that can be directly implemented.
- A **procedural abstraction** refers to a sequence of instructions that have a specific and limited function.
- A **data abstraction** is a named collection of data that describes a data object.

Design Concepts

- **Architecture**
  - Software architecture - the overall structure of the software and the ways in which that structure provides conceptual integrity for a system.

- **Patterns**
  - The intent of each design pattern is to provide a description that enables a designer to determine
  - (1) whether the pattern is applicable to the current work,
  - (2) whether the pattern can be reused
  - (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

- **Separation of Concerns**

- Modularity
- Information Hiding

Design Concepts

➢ Separation of Concerns

- ***Separation of concerns*** is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.
- **A *concern*** is a feature or behavior that is specified as part of the requirements model for the software.

➢ Modularity

- Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called *modules*, that are integrated to satisfy problem requirements.

➢ Information Hiding

Design Concepts

- Functional Independence
- Refinement
- Aspects
- Refactoring
- Object-Oriented Design Concepts
- Design Classes

Design Model

1. Data Design Elements
2. Architectural Design Elements
3. Interface Design Elements
4. Component-Level Design Elements
5. Deployment-Level Design Elements

Software Quality Concepts

1. What Is Quality?

2. Software Quality
    - Garvin’s Quality Dimensions
    - McCall’s Quality Factors
    - ISO 9126 Quality Factors
    - Targeted Quality Factors
    - The Transition to a Quantitative View

Software Quality Concepts

3. The Software Quality Dilemma
    - “Good Enough” Software
    - The Cost of Quality
    - Risks
    - Negligence and Liability
    - Quality and Security
    - The Impact of Management Actions

Software Quality Concepts

4. Achieving Software Quality
    - Software Engineering Methods
    - Project Management Techniques
    - Quality Control
    - Quality Assurance

Review Metrics

Review Metrics

- Analyzing Metrics
- Cost Effectiveness of Reviews


Thank You!