Unit 2: Requirements Modelling and Design

Dr Sagar Deep Deb, PhD

 # Requirements Engineering

Requirements Engineering is the process of finding out what the customer really wants, writing it clearly, and making sure it is correct and manageable. To guarantee the effective creation of a software product, the requirements engineering process entails several tasks that help in understanding, recording, and managing the demands of stakeholders.

 # Requirements Engineering Process

1. **Feasibility Study**: It asks if it possible/feasible to build this system. It is broadly divided into the following categories:
   (i) Technical Feasibility: In Technical Feasibility current resources both hardware software along required technology are analysed/assessed to develop the project. Along with this, the feasibility study also analyses the technical skills and capabilities of the technical team.
   (ii) Operational Feasibility: In Operational Feasibility degree of providing service to requirements is analysed along with how easy the product will be to operate and maintain after deployment. In short will the users accept it?
   (iii) Economic Feasibility: In the Economic Feasibility study cost and benefit of the project are analysed.
   (iv) Legal Feasibility: In legal feasibility, the project is ensured to comply with all relevant laws, regulations, and standards. It identifies any legal constraints that could impact the project and reviews existing contracts and agreements to assess their effect on the project's execution. Additionally, legal feasibility considers issues related to intellectual property, such as patents and copyrights, to safeguard the project's innovation and originality.
   (v) Schedule/Time Feasibility: In schedule feasibility, the project timeline is evaluated to determine if it is realistic and achievable.
2. **Requirements Elicitation:** In this step the question asked is what does the user actually want? This step is about collecting requirements from stakeholders. It is related to the various ways used to gain knowledge about the project domain and requirements. The various sources of domain knowledge include customers, business manuals, the existing software of the same type, standards, and other stakeholders of the project. Common methods: Interviews, Questionnaires, Meetings/workshops, Observation, Prototyping.
3. **Requirements Specification:** The question asked here is are how do we write requirements clearly? All analyzed requirements are written in a formal document called Software Requirements Specification (SRS). This activity is used to produce formal software requirement models. All the requirements including the functional as well as the non-functional requirements and the constraints are specified by these models in totality. During specification, more knowledge about the problem may be required which can again trigger the elicitation process. The models used at this stage include ER diagrams, data flow diagrams(DFDs), function decomposition diagrams(FDDs), data dictionaries, etc.
   Several types of requirements are commonly specified in this step, including
   Functional Requirements: These describe what the software system should do. They specify the functionality that the system must provide, such as input validation, data storage, and user interface. It specifically answers what should the system do when a user interacts with it?
   Non-Functional Requirements: These describe how well the software system should do it. They specify the quality attributes of the system, such as performance, reliability, usability, and security.
   Example: In a University ERP system, functional requirements include allowing students to register for courses, enabling faculty to upload marks, allowing administrators to generate reports, and sending system notifications. Non-functional requirements specify that the system should handle a large number of students simultaneously, ensure confidentiality of academic data, provide fast response time, and maintain high availability and reliability.
   Constraints: These describe any limitations or restrictions that must be considered when developing the software system.

Acceptance Criteria: These describe the conditions that must be met for the software system to be considered complete and ready for release.

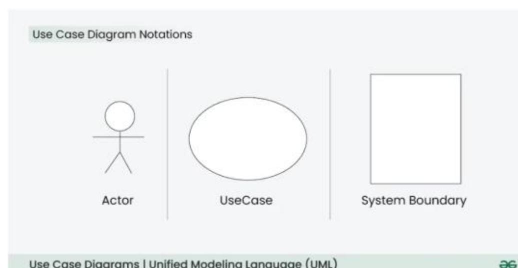
4. **Requirements Verification and Validation Verification**: **Question:** *Did we understand the user correctly?* It refers to the set of tasks that ensures that the software correctly implements a specific function. Validation: It refers to a different set of tasks that ensures that the software that has been built is traceable to customer requirements.
5. **Requirements Management** Question: How do we handle changes? Requirements often change due to: New user needs Market changes Policy changes This step ensures: Changes are tracked Old versions are maintained Impact of changes is analyzed.

 # Real life example Attendance Management System at UPES.

In an Attendance Management System, requirements engineering starts with a feasibility study to check technical, economic, and operational viability. Requirements are then elicited from teachers, students, and administrators to understand how attendance should be recorded and managed. These requirements are clearly documented in a Software Requirements Specification (SRS), including functional requirements like marking attendance and generating reports, and non-functional requirements such as security and system performance. Verification and validation ensure that the system meets user expectations. Requirements management handles future changes such as adding biometric or mobile-based attendance.

 # Use case Diagrams

A Use Case Diagram is a visual way to show how users (actors) interact with a system and what functions (use cases) the system provides. It helps understand the system's behavior from the user's perspective. Represents actors (users or external systems) and use cases (system functionalities). Shows the relationships between users and system features. Helps define system scope, requirements, and interactions clearly.



Actors are external entities that interact with the system. These can include users, other systems, or hardware devices. In the context of a Use Case Diagram, actors initiate use cases and receive the outcomes.

Use cases are like scenes in the play. They represent specific things your system can do. In the online shopping system, examples of use cases could be "Place Order," "Track Delivery," or "Update Product Information".

The system boundary is a visual representation of the scope or limits of the system you are modeling. It defines what is inside the system and what is outside. The boundary helps to establish a clear distinction between the elements that are part of the system and those that are external to it.

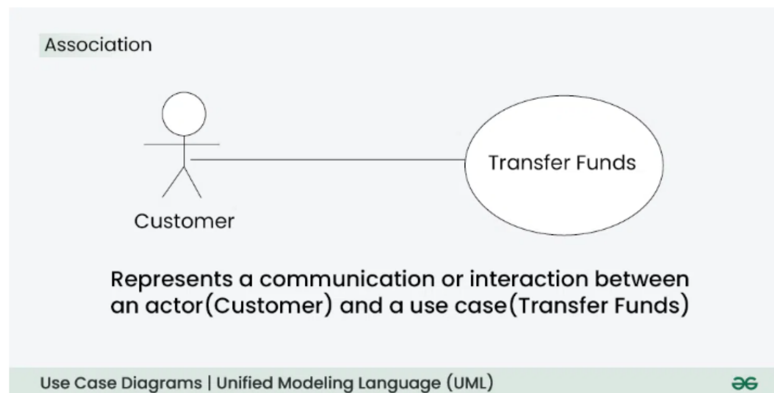
 # Use case Relationships

1. Association Relationship

The Association Relationship represents a communication or interaction between an actor and a use case. It is depicted by a line connecting the actor to the use case. This relationship signifies that the actor is involved in the functionality described by the use case.

Example: Online Banking System

- **Actor:** Customer
- **Use Case:** Transfer Funds
- **Association:** A line connecting the "Customer" actor to the "Transfer Funds" use case, indicating the customer's involvement in the funds transfer process.

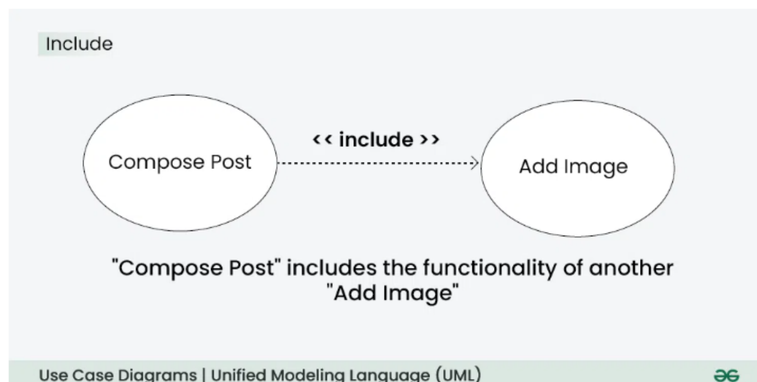


2. Include Relationship

The Include Relationship indicates that a use case includes the functionality of another use case. It is denoted by a dashed arrow pointing from the including use case to the included use case. This relationship promotes modular and reusable design.

Example: Social Media Posting

- **Use Cases:** Compose Post, Add Image
- **Include Relationship:** The "Compose Post" use case includes the functionality of "Add Image." Therefore, composing a post includes the action of adding an image.

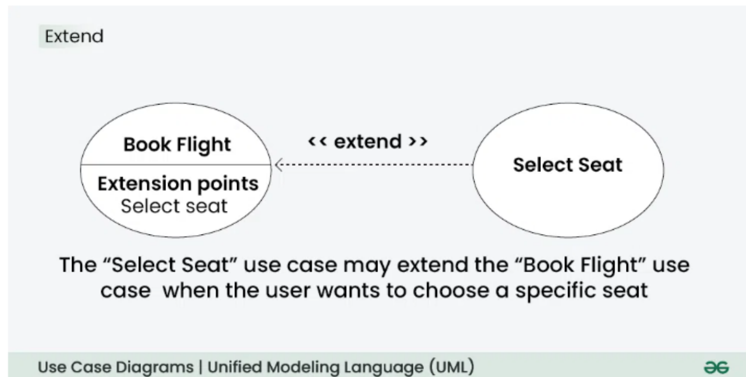


3. Extend Relationship

The Extend Relationship illustrates that a use case can be extended by another use case under specific conditions. It is represented by a dashed arrow with the keyword "extend." This relationship is useful for handling optional or exceptional behavior.

> ***Example: Flight Booking System***

- **Use Cases:** Book Flight, Select Seat
- **Extend Relationship:** The "Select Seat" use case may extend the "Book Flight" use case when the user wants to choose a specific seat, but it is an optional step.

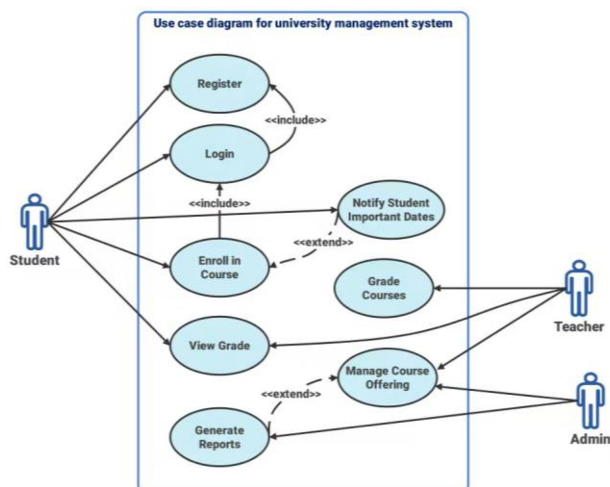


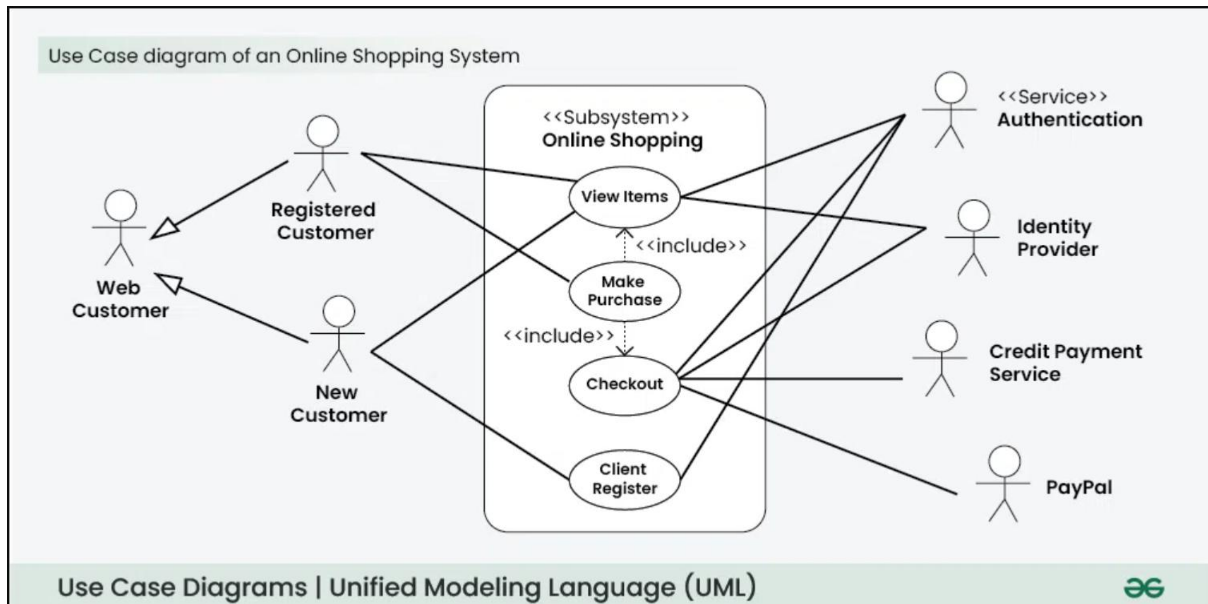
4. Generalization Relationship

The Generalization Relationship establishes an "is-a" connection between two use cases, indicating that one use case is a specialized version of another. It is represented by an arrow pointing from the specialized use case to the general use case.

> ***Example: Vehicle Rental System***

- **Use Cases:** Rent Car, Rent Bike
- **Generalization Relationship:** Both "Rent Car" and "Rent Bike" are specialized versions of the general use case "Rent Vehicle."





In this use case diagram of an online shopping system, each section and line type has a specific meaning: the stick figures represent actors (such as Web Customer, Registered Customer, Authentication Service, PayPal), which are external users or systems interacting with the software; the large rectangle labeled Online Shopping is the system boundary, showing what functionalities belong to the system; the ovals inside the boundary are use cases (View Items, Make Purchase, Checkout, Client Register), representing the services the system provides. The solid lines between actors and use cases indicate direct interaction or association, meaning that the actor can perform or participate in that use case. The dashed arrows labeled «include» represent an include relationship, meaning one use case always depends on another—for example, Make Purchase includes View Items and Checkout, because purchasing cannot occur without viewing items and checking out. External service actors like Authentication, Credit Payment Service, and PayPal are connected with solid lines to show system integration for specific functions such as login verification and payment processing. Overall, solid lines show who interacts with what, while dashed lines show mandatory functional dependency between use cases.