

Chapter 3

Requirements Engineering and Principles

3.1 Requirements and its types

Requirements in software engineering are the specifications that define what a software system should do and the constraints under which it should operate. They serve as the foundation for system design and development, ensuring that the final product meets the needs of its users and stakeholders.

Types of Requirements

There are broadly two types of requirements:

1. **Functional Requirements:** Describe what the system should do (services, behaviors, and functionality). These are user-centered features such as login processes, data manipulation, and report generation.

Examples:

- The system must allow users to register and log in.
- The system should process payments through multiple gateways.
- The system should generate monthly reports of sales data.

2. **Non-functional Requirements:** Non-functional requirements describe **how the system should perform** rather than what it does. They address quality attributes, system constraints, and standards. They specify how the system should perform, covering aspects like usability, performance, security, scalability, and maintainability.

Examples:

- **Performance:** The system should handle 10,000 concurrent users.
- **Scalability:** The system should be able to scale to accommodate increased user traffic.
- **Reliability:** The system should have 99.9% uptime.
- **Security:** User passwords must be stored securely using encryption.
- **Usability:** The system should be intuitive for first-time users.
- **Maintainability:** The codebase should allow for easy updates.

However, we have many more other types of requirements beside functional and non-functional requirements.

1. User Requirements

- **Definition:** User requirements focus on what **end-users** need and expect from the system. These are often documented as **user stories** or **use cases**.
- **Examples:**
 - Users should be able to retrieve their account information easily.
 - Users should be able to customize their settings.
 - As a customer, I want to track my order status online.
 - As an administrator, I want to manage user roles and permissions.

2. System Requirements

- **Definition:** System requirements provide a detailed technical description of the hardware and software necessary to implement the system.
- **Examples:**
 - The system must run on Windows, macOS, and Linux
 - The system requires a minimum of 16 GB of RAM and 500 GB of storage.
 - The system should be compatible with Windows and macOS operating systems.'

3. Business Requirements

- **Definition:** Business requirements outline the **high-level objectives** and goals of the organization that the software must meet.
- **Examples:**
 1. Increase sales by 20% through improved online shopping experiences.
 2. Reduce customer support response times to enhance customer satisfaction.

Importance of Requirements

Requirements in **software engineering** are critically important because they form the foundation for the development, implementation, and evaluation of a software system. Here's a detailed look at their importance:

1. **Clarify the Problem:** Ensure understanding of what needs to be solved.
2. **Guide Design and Development:** Act as a blueprint for building the system.
3. **Align with Stakeholder Expectations:** Ensure the software delivers value.
4. **Enable Accurate Estimation:** Help plan resources, time, and budget.
5. **Facilitate Communication:** Serve as a common language among teams.
6. **Support Testing and Validation:** Provide benchmarks for ensuring functionality.
7. **Reduce Risks:** Identify potential issues early for mitigation.
8. **Aid Maintenance and Upgrades:** Serve as a reference for future work.
9. **Ensure Compliance:** Address legal and regulatory standards.
10. **Improve Quality:** Lead to robust, reliable, and user-friendly software.

3.2 Requirements Modeling Principles

Requirement modeling principles provide a structured approach to ensure that software requirements are properly understood, documented, and translated into system design. These principles help in creating accurate, clear, and effective models that capture both functional and non-functional requirements. The following are the key principles:

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

- Identify and model the data and information that the system processes, ensuring clarity about what the software will handle.

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

- Clearly specify what the software is supposed to do, including its tasks, computations, and operations.

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

- Model how the software responds to external inputs or events, ensuring its dynamic aspects are understood.

Principle 4. The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion

- Break down the models into layers or hierarchies, starting with a high-level overview and progressively adding detailed specifics.

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

- Start by modeling the core requirements without technical details, then gradually refine the models to include implementation-specific details.

3.3 Domain Analysis and System Models

Domain Analysis:

Definition:

Domain analysis is the process of studying a specific problem area to identify common features, requirements, and reusable components that can improve software development.

Domain analysis is the process of identifying, analyzing, and modeling the **commonalities and variabilities** within a specific domain of interest to create reusable components and facilitate efficient software development.

Objective: To understand the domain and identify reusable patterns, components, and frameworks that can reduce development time and effort.

Example of Domain Analysis:

In an e-commerce domain, common entities might include products, customers, orders, and payments. Domain analysis would identify reusable components like shopping cart functionality, product catalogs, and payment gateways, which can be applied to different e-commerce platforms.

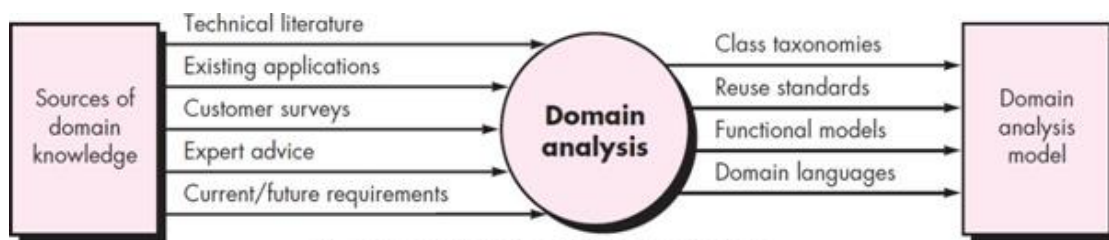


Fig. Input and output for domain analysis

Steps in Domain Analysis:

1. **Identify the Domain:** Define the scope and boundaries of the domain (e.g., healthcare, e-commerce).
2. **Gather Domain Knowledge:** Collect information from existing systems, literature, and domain experts.
3. **Identify Common Features:** Determine the features and requirements common to most applications in the domain.

4. **Analyze Variations:** Understand where and why differences exist between systems.
5. **Develop Domain Models:** Create models to represent domain concepts, processes, and relationships.

Outputs of Domain Analysis:

1. **Domain Model:** A conceptual representation of the domain, including entities, relationships, and rules.
2. **Reusable Assets:** Components, code, or patterns that can be reused in multiple applications

Techniques:

- **Interviews:** Gather insights from domain experts.
- **Surveys:** Collect broader input on user needs.
- **Document Analysis:** Review existing materials for historical context.
- **Workshops:** Facilitate collaborative discussions among stakeholders.

Importance:

- Encourages reuse of software components, reducing costs and development time.
- Improves consistency across similar applications.
- Facilitates better understanding of the domain for developers and stakeholders.

System modeling:

System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. System modeling has generally come to mean representing the system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML).

System models are essential tools used during the requirements engineering and system design phases. These models help to represent various aspects of a software system, providing a clear, structured view of its behavior, structure, and interaction with external systems.

Models are used during the requirements engineering process to help derive the requirements for a system, during the design process to describe the system to engineers implementing the system and after implementation to document the system's structure and operation.

Purpose of System Models:

1. **Clarification and Communication:** System models provide a means to clearly represent the system's behavior, structure, and interfaces, which helps both developers and stakeholders understand the requirements and design of the system.
2. **Problem Representation:** They allow complex systems to be understood more easily by breaking them down into manageable parts and providing an abstract view of the system that is independent of implementation details.
3. **Requirements Validation:** Models can be used to validate requirements by making sure that the system meets the user's needs, and they help in identifying inconsistencies, gaps, or ambiguities in the requirements.

Types of System Models:

1. **Context model:** A **context model** defines the boundaries of a system by showing its interactions with external entities, such as users, other systems, or devices. It helps identify what is **inside** and **outside** the system, focusing on **external interfaces** and **high-level interactions**.
2. **Behavioral Model:** A **behavioral model** describes how a system behaves in response to external events or stimuli. It focuses on the **dynamic aspects** of the system, showing how it reacts to inputs, changes states, and performs actions. Examples include state diagrams and sequence diagrams.
3. **Dataflow Model:** A **dataflow model** shows how **data moves** through the system and how it is processed by different components. It represents the flow of data between various system processes or functions, illustrating inputs, outputs, and transformations in a **functional** manner.
4. **Structural Model:** A **structural model** defines the **static organization** of a system, describing its components, their relationships, and how they are organized. It focuses on the

system's architecture and the structure of **subsystems, classes, and modules**. Examples include class diagrams and component diagrams.

5. **Interaction Model:** An **interaction model** focuses on how the different components or entities within the system **interact** with each other. It describes **communication protocols, sequences of actions**, or messages exchanged between system components. Examples include sequence diagrams, collaboration diagrams, and communication diagrams.

3.3.1 Context Models

Definition: A **Context Model** (or Context Diagram) is a high-level visual representation of a system and its interactions with external entities. It helps define the system's boundaries and identify how it interacts with users and other systems.

Purpose

- **Clarify System Boundaries:** Distinguishes what is inside and outside the system.
- **Identify External Entities:** Shows all stakeholders, users, and other systems that interact with the system.
- **Visualize Data Flows:** Represents the inputs and outputs exchanged between the system and its environment.

Key Components:

1. **System:** The central box or a circle representing the system being modeled.
2. **External Entities:** Rectangles or ovals around the system that represent users, other systems, or organizations.
3. **Data Flows:** Arrows indicating the flow of information between the system and external entities.

Steps to Create a Context Model:

1. **Identify the System:** Define the primary system you are modeling.
2. **Identify External Entities:** List all actors that will interact with the system (e.g., users, other software).
3. **Define Interactions:** Specify how each external entity interacts with the system and what data is exchanged.
4. **Draw the Diagram:** Represent the system, external entities, and data flows visually.

Example:

- **Library Management System:**
 - **External Entities:**
 - **User:** Requests to borrow or return books.
 - **Book Database:** Provides book information.
 - **Notification System:** Sends overdue alerts.
 - **Data Flows:** Arrows show interactions like requests and responses.

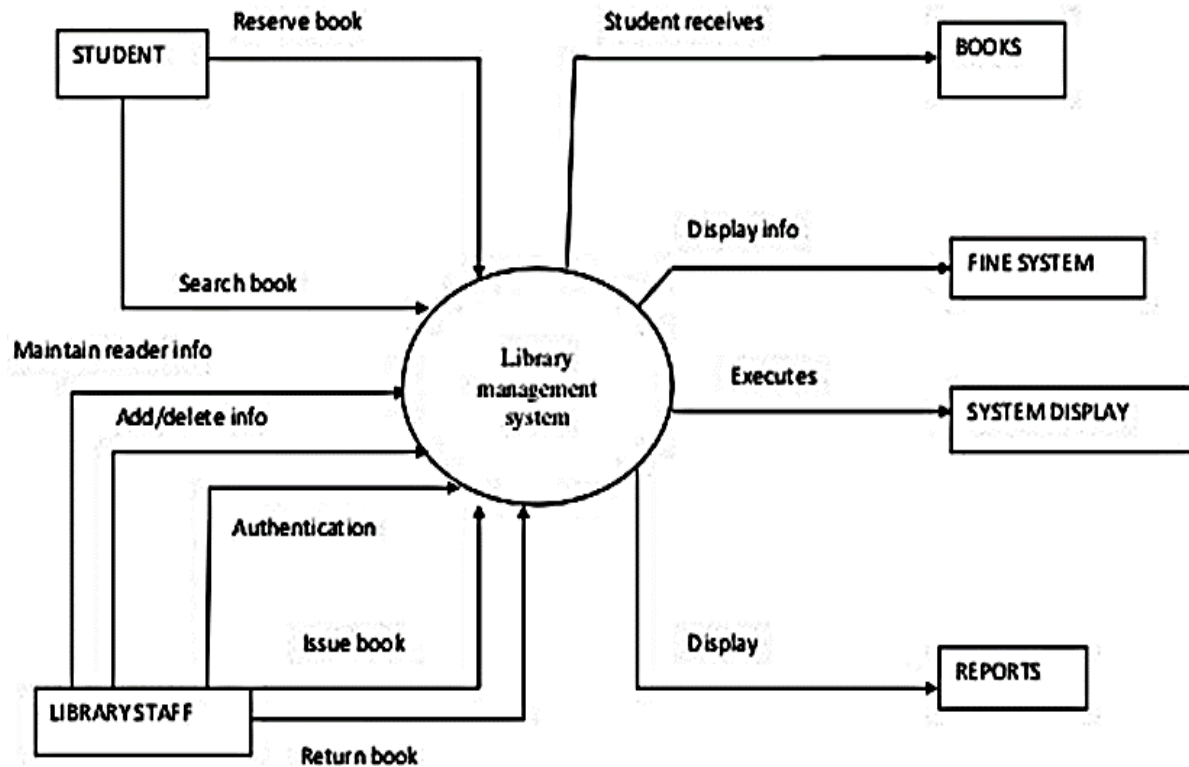


Figure: Context Diagram for a Library Management System

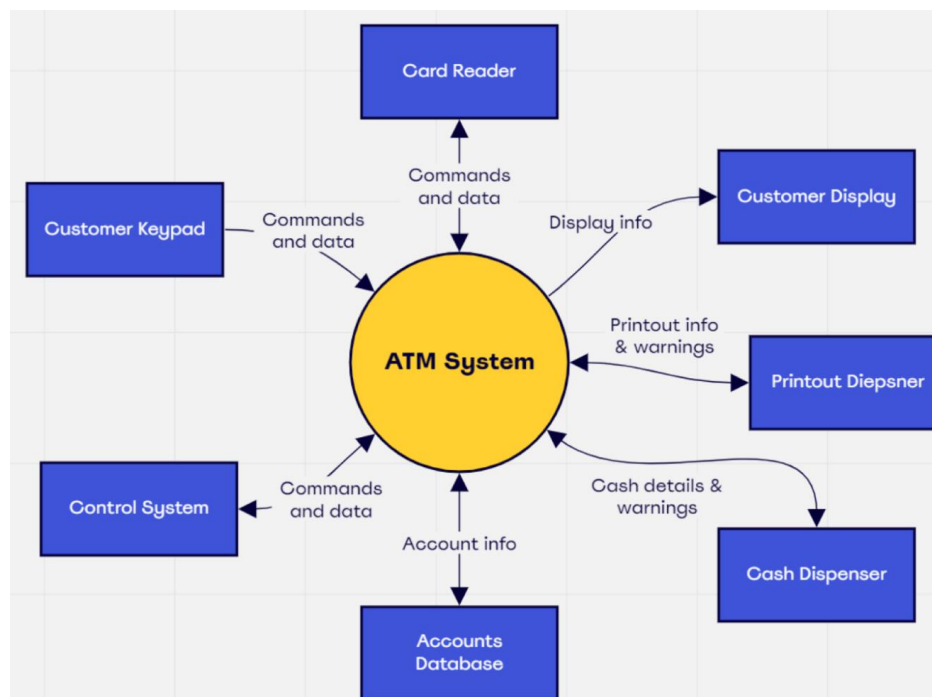


Figure: Context diagram for ATM system

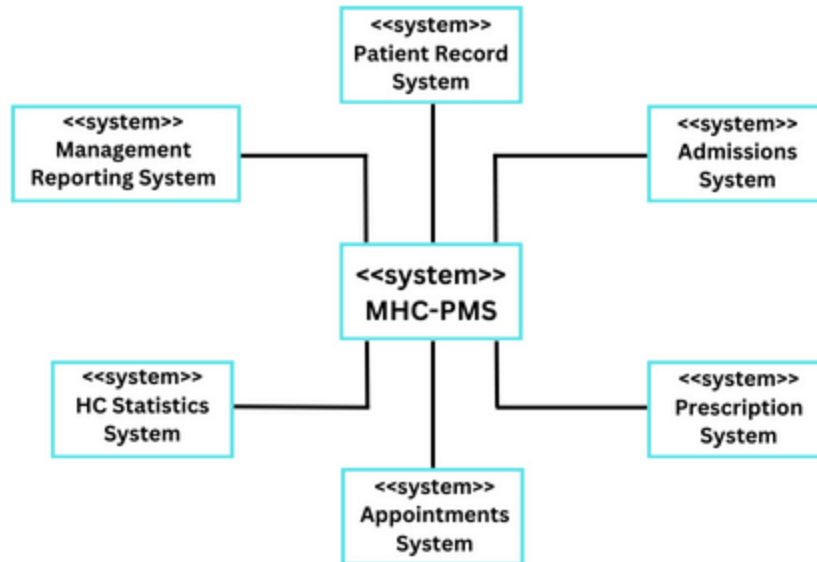


Figure: Context model of the MHC-PMS

The diagram below shows the main system at the centre, **MHC-PMS (Mental Health Care Patient Management System)**. This system can be used in clinics and help them maintain patients' records. It comprises various sub-systems: the **Patient Record System**, **Admission System**, **Prescription System**, **Appointments System**, **HC Statistics System**, and **Management Reporting System**. These sub-systems are connected to the main system. These sub-systems support and work with the main system.

3.3.2 Behavioral Models

Behavioral models are representations that illustrate how a system behaves in response to internal and external events. They focus on the dynamic aspects of a system, showing how it responds over time to various inputs and conditions.

Purpose:

- **Understand System Behavior:** Helps in analyzing how the system reacts under different scenarios.
- **Define Interactions:** Clarifies the interactions between different components of the system.
- **Support Requirements Gathering:** Assists in identifying and refining functional requirements based on behavioral insights.

Key Types of Behavioral Models

1. Use Case Diagrams:	<ul style="list-style-type: none">○ Visual representation of system functionalities from a user perspective.○ Shows interactions between users (actors) and the system through various use cases.
2. Activity Diagrams:	<ul style="list-style-type: none">○ Represents workflows of stepwise activities and actions.○ Useful for modeling business processes and system workflows.
3. Sequence Diagrams:	<ul style="list-style-type: none">○ Depicts interactions between objects in a time sequence.○ Shows how processes operate with one another and in what order.
4. State Diagrams:	<ul style="list-style-type: none">○ Illustrates the states an object can be in and the transitions between these states based on events.○ Useful for modeling dynamic behavior and lifecycle of objects.

1. Use case Diagrams

- A Use Case Diagram is a visual representation of the functional requirements of a system.
- It describes the interactions between users (or other systems) and the system itself, focusing on *what* the system should do, rather than *how* it does it.
- Use case diagrams are a core part of the Unified Modeling Language (UML) and are widely used in software engineering to capture and communicate requirements.

Notations used:

1. Actors:

- Represent the users or external systems that interact with the system.
- Denoted as stick figures.
- Types:
 - Primary Actors: Directly use the system to achieve a goal.
 - Secondary Actors: Assist in completing the use cases.

2. Use Cases:

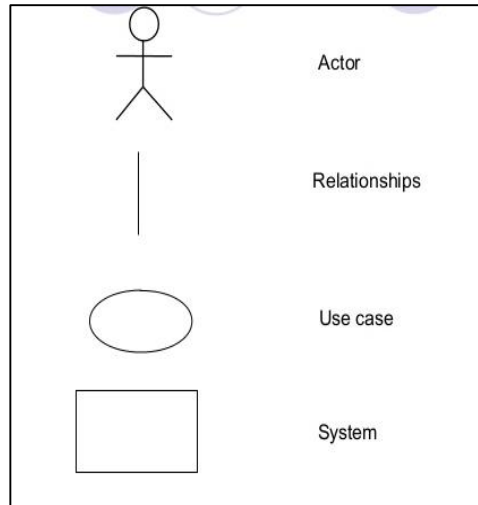
- Represent specific functionalities or goals of the system from the user's perspective.
- Depicted as ovals.
- Example: "Login to the system," "Place an order," etc.

3. System Boundary:

- Represents the scope of the system.
- Shown as a rectangle enclosing the use cases.

4. Relationships:

- Association: A link between actors and use cases indicating interaction.
- Include: Indicates a use case is mandatory and reused within another use case.
- Extend: Indicates optional or conditional behavior that extends the base use case.
- Generalization: Indicates inheritance between actors or use cases.



The purposes of Use-Case are:

1. To identify system functionality as seen by external users.
2. To serve as a communication tool between stakeholders, developers, and analysts.
3. To clarify system boundaries and scope.
4. To act as a foundation for further modeling or development activities, like activity or sequence diagrams.

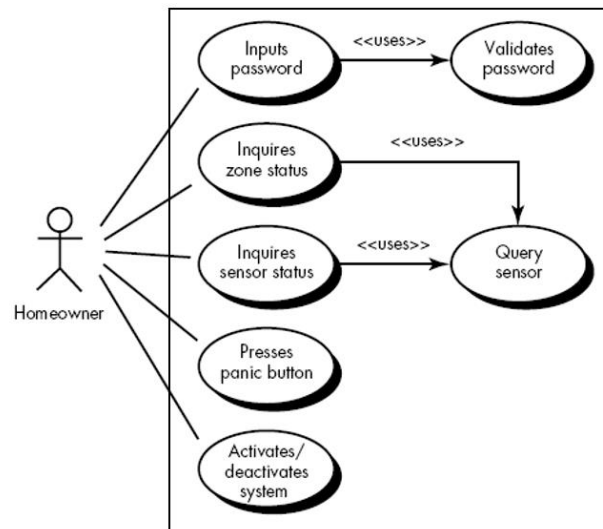


Figure: Usecase diagram for Safe Home System

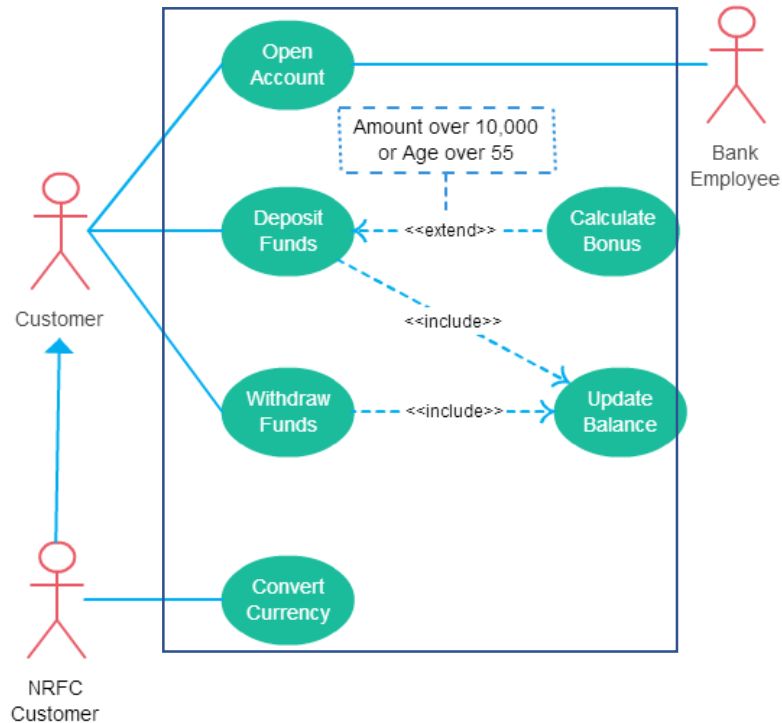






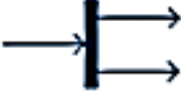

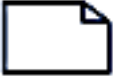


Figure: Use case diagram for a simple banking system

2. Activity Diagram

Activity diagram is basically a flowchart to represent the flow from one activity to another activity. The activity can be described as an operation of the system. The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario.

Symbols used:

Symbol	Name	Use
	Start/ Initial Node	Used to represent the starting point or the initial state of an activity
	Activity / Action State	Used to represent the activities of the process
	Control Flow / Edge	Used to represent the flow of control from one action to the other
	Activity Final Node	Used to mark the end of all control flows within the activity
	Decision Node	Used to represent a conditional branch point with one input and multiple outputs
	Merge Node	Used to represent the merging of flows. It has several inputs, but one output.
	Fork	Used to represent a flow that may branch into two or more parallel flows
	Merge	Used to represent two inputs that merge into one output
	Note/ Comment	Used to add relevant comments to elements

Example:

Given the problem description related to the workflow for processing an order, let's model the description in visual representation using an activity diagram:

Process Order - Problem Description

Once the order is received, the activities split into two parallel sets of activities. One side fills and sends the order while the other handles the billing.

On the Fill Order side, the method of delivery is decided conditionally. Depending on the condition either the Overnight Delivery activity or the Regular Delivery activity is performed.

Finally the parallel activities combine to close the order.

The activity diagram example below visualizes the flow in graphical form

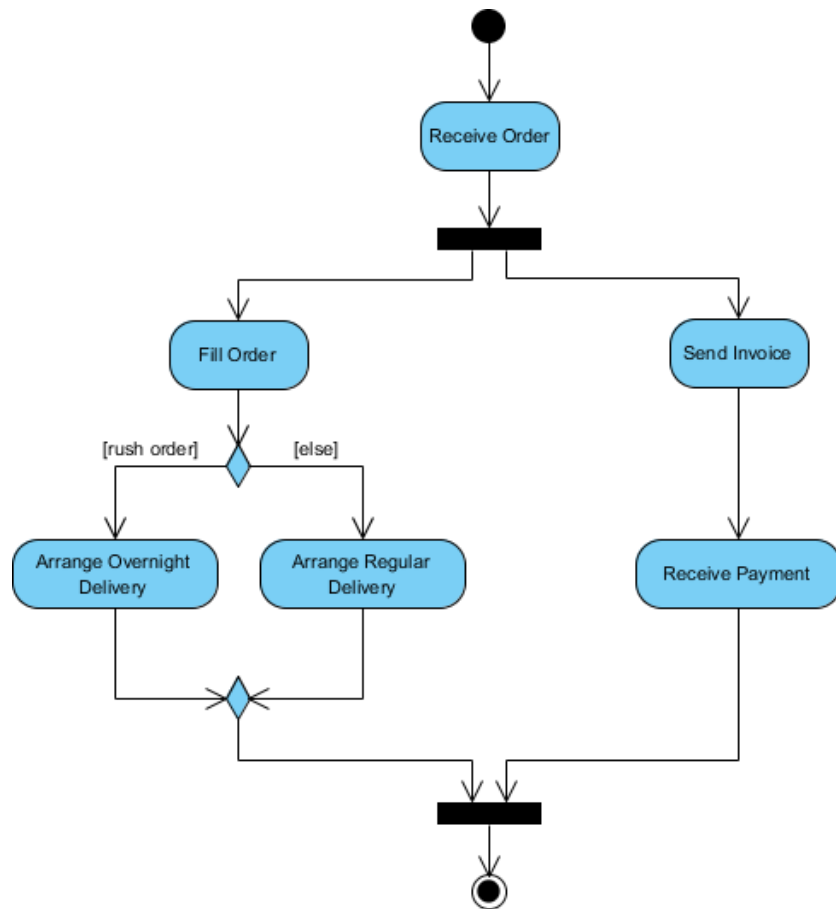
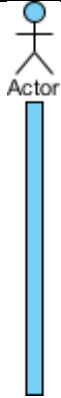

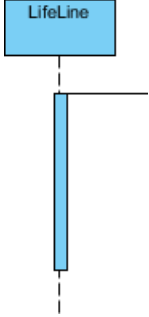
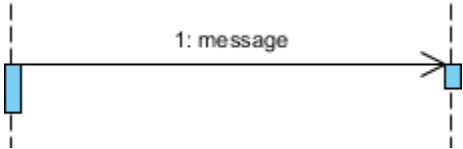


Figure: Activity diagram for an Order Management System

3. Sequence Diagram

A *sequence diagram* is an interaction diagram that emphasizes the time ordering of messages. It shows a set of objects and the messages sent and received by those objects. Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis.

Symbols used:

Notation Description	Visual Representation
Actor <ul style="list-style-type: none"> a type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data) external to the subject (i.e., in the sense that an instance of an actor is not a part of the instance of its corresponding subject). represent roles played by human users, external hardware, or other subjects. <p>Note that:</p> <ul style="list-style-type: none"> An actor does not necessarily represent a specific physical entity but merely a particular role of some entity A person may play the role of several different actors and, conversely, a given actor may be played by multiple different person. 	
Lifeline <ul style="list-style-type: none"> A lifeline represents an individual participant in the Interaction. 	
Activations <ul style="list-style-type: none"> A thin rectangle on a lifeline) represents the period during which an element is performing an operation. The top and the bottom of the of the rectangle are aligned with the initiation and the completion time respectively 	
Call Message <ul style="list-style-type: none"> A message defines a particular communication between Lifelines of an Interaction. 	

<ul style="list-style-type: none"> Call message is a kind of message that represents an invocation of operation of target lifeline. 	
Return Message <ul style="list-style-type: none"> A message defines a particular communication between Lifelines of an Interaction. Return message is a kind of message that represents the pass of information back to the caller of a corresponded former message. 	
Self Message <ul style="list-style-type: none"> A message defines a particular communication between Lifelines of an Interaction. Self message is a kind of message that represents the invocation of message of the same lifeline. 	
Recursive Message <ul style="list-style-type: none"> A message defines a particular communication between Lifelines of an Interaction. Recursive message is a kind of message that represents the invocation of message of the same lifeline. It's target points to an activation on top of the activation where the message was invoked from. 	
Create Message <ul style="list-style-type: none"> A message defines a particular communication between Lifelines of an Interaction. Create message is a kind of message that represents the instantiation of (target) lifeline. 	
Destroy Message <ul style="list-style-type: none"> A message defines a particular communication between Lifelines of an Interaction. Destroy message is a kind of message that represents the request of destroying the lifecycle of target lifeline. 	
Duration Message <ul style="list-style-type: none"> A message defines a particular communication between Lifelines of an Interaction. Duration message shows the distance between two time instants for a message invocation. 	

Note

- A note (comment) gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler.

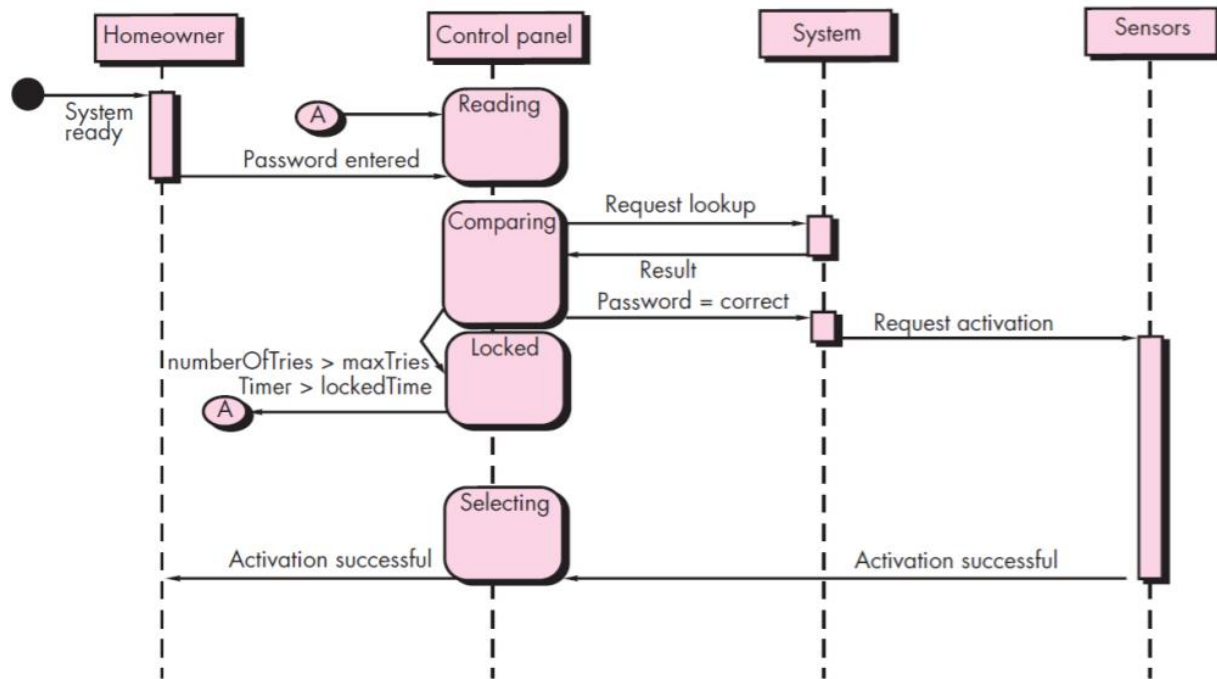


Figure: Sequence Diagram for the SafeHome Security function

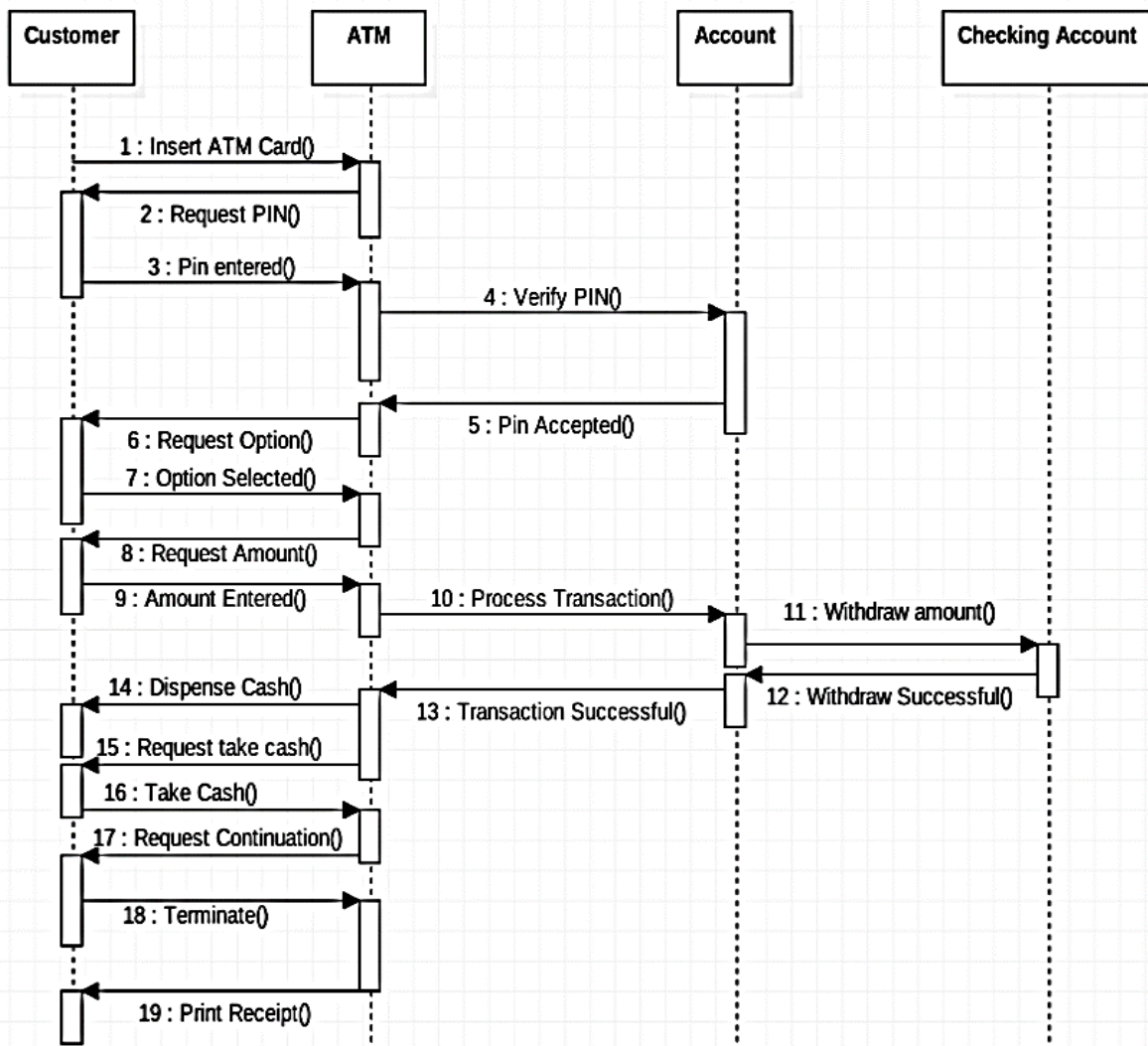






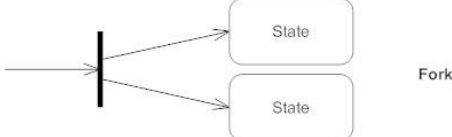
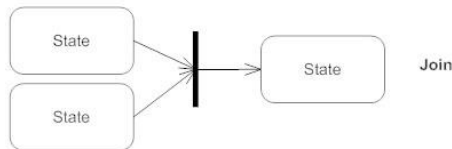


Figure: Sequence diagram for an ATM

4. State Diagram

State chart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. State chart diagram is one of the UML diagrams used to model the dynamic nature of a system. They define different states of an object during its lifetime and these states are changed by events.

Basic State Chart Diagram Symbols and Notations

Details	Graphical notation
States: States represent situations during the life of an object. You can easily illustrate a state by using a rectangle with rounded corners.	 A simple state  A state with internal activities
Transition: A solid arrow represents the path between different states of an object. Label the transition with the event that triggered it and the action that results from it. A state can have a transition that points back to itself.	 Transition
Self transition: We use a solid arrow pointing back to the state itself to represent a self transition. There might be scenarios when the state of the object does not change upon the occurrence of an event. We use self transitions to represent such cases.	
Initial State: A filled circle followed by an arrow represents the object's initial state.	 Initial state
Final State: An arrow pointing to a filled circle nested inside another circle represents the object's final state.	 Final state
Synchronization and Splitting of Control: A short heavy bar with two transitions entering it represents a synchronization of control. The first bar is often called a fork where a single transition splits into concurrent multiple transitions. The second bar is called a join, where the concurrent transitions reduce back to one.	 Fork  Join

Examples:

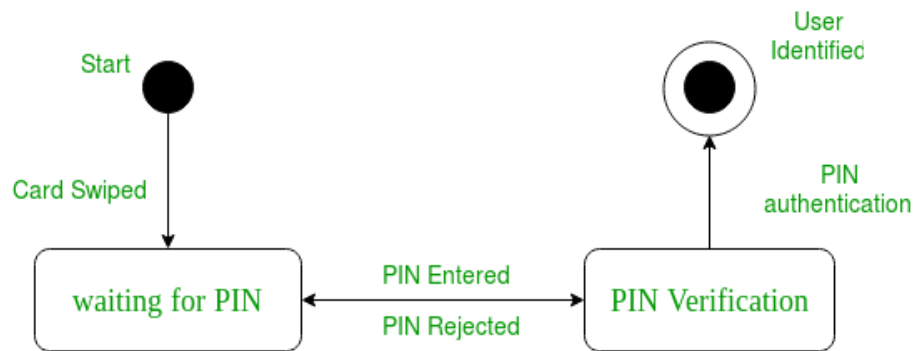


Figure: A state diagram for User Verification

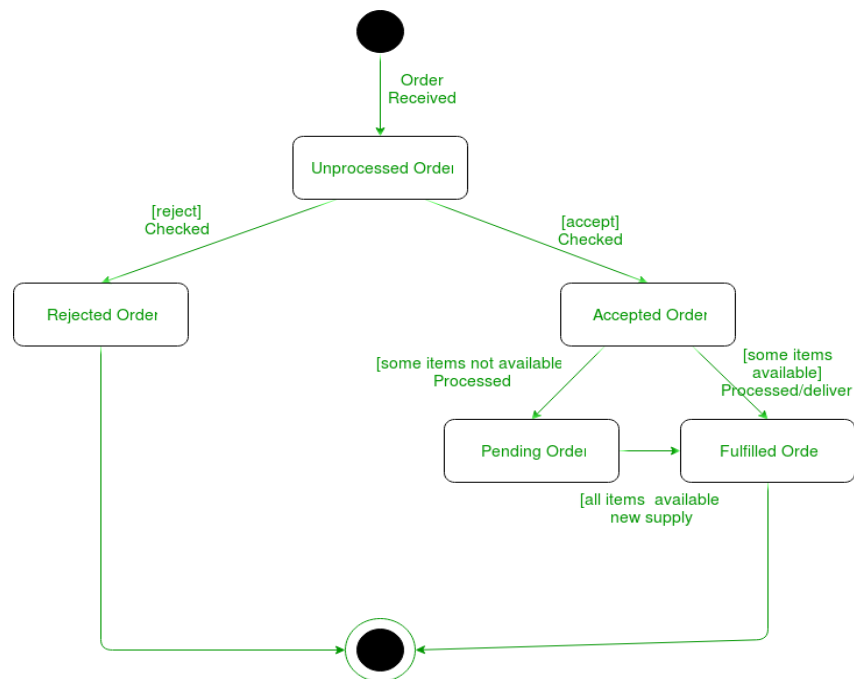


Figure: A State Diagram for an online order

3.3.3 Data Models

A **data model** is a conceptual representation of data structures and relationships within a system. It defines how data is stored, accessed, and manipulated, serving as a blueprint for database design and information management. It provides a systematic way to visualize data elements and their relationships, guiding the development of databases and information systems.

Components of Data Models

- **Entities:** Objects or concepts that represent real-world items (e.g., Student, Course).
- **Attributes:** Characteristics or properties of entities (e.g., Student_ID, Course_Name).
- **Relationships:** Links between entities that show how they interact (e.g., a Student enrolls in a Course).
- **Constraints:** Rules that enforce data integrity, such as primary key constraints and foreign key relationships.

Entity Relationship diagram (ERD)

The ERD was originally proposed by Peter Chen for the design of relational database systems and has been extended by others. A set of primary components are identified for the ERD: data objects, attributes, relationships, and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships (data object type hierarchies and associative data object). ERD uses graphical notation, three basic elements in ER models are:

- Entities are the "things" about which we seek information (**Rectangle**).
- Attributes are the data we collect about the entities (**Oval**).
- Relationships provide the structure needed to draw information from multiple entities (Line and Diamond).

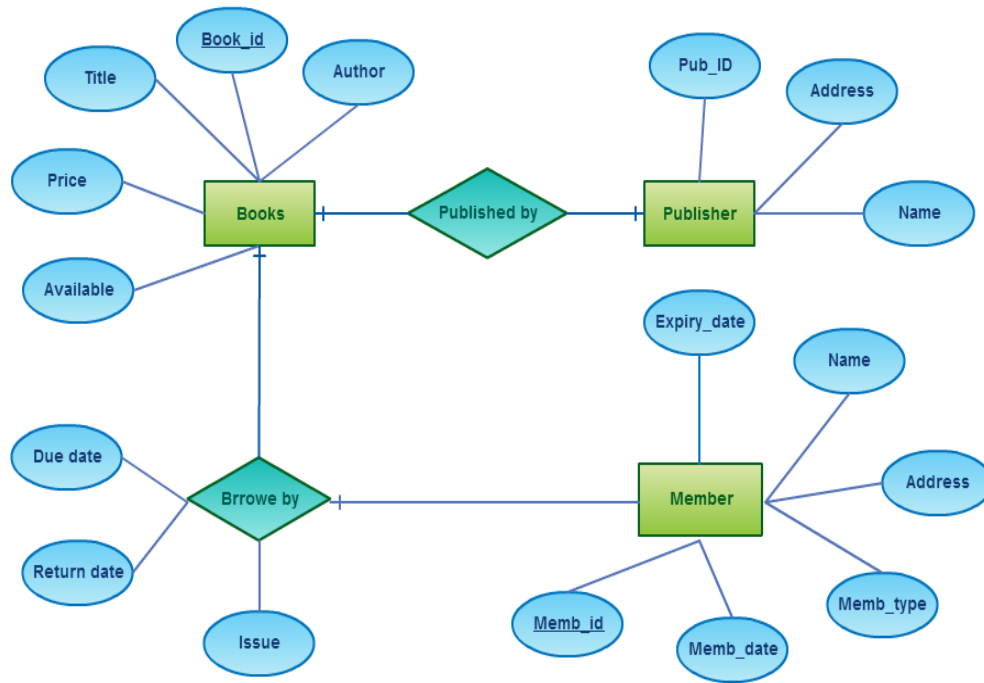


Figure: ERD of Library Management System

3.3.4 Structural Models

Structural modeling in software engineering focuses on representing the static aspects of a system — specifically how data and components are organized, how they relate to each other, and how they are structured within the system. The goal of structural modeling is to capture the system's architecture and organization, providing a blueprint for how the system's components interact and are structured.

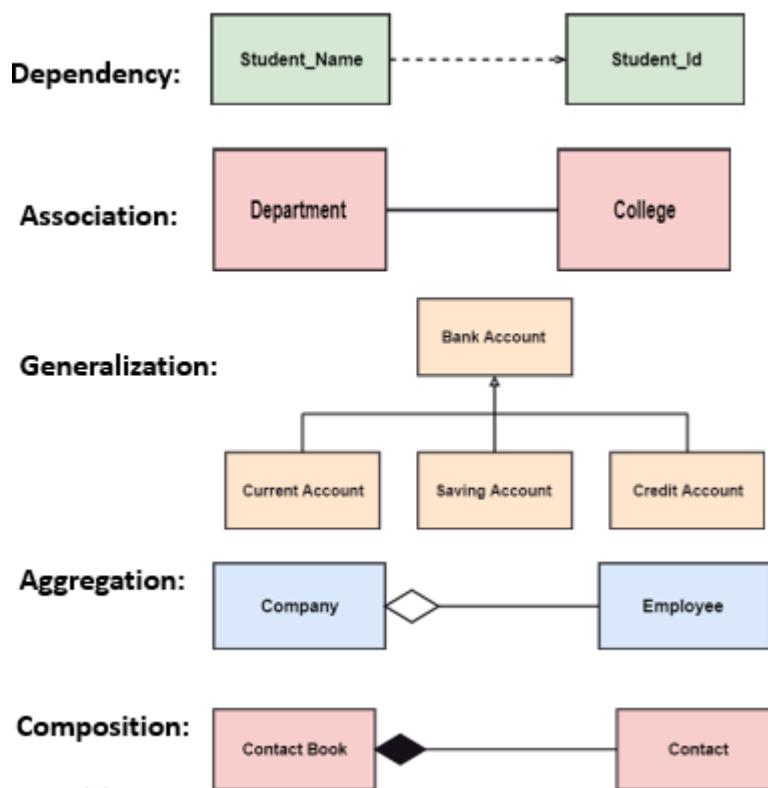
Types of structural models:

1. Class diagram	<ul style="list-style-type: none"> Represents the classes of a system and their attributes, operations, and the relationships between them (such as inheritance, association, and dependency).
2. Object Diagram	<ul style="list-style-type: none"> Focuses on instances of classes (objects) and their relationships at a specific point in time. Object diagrams are similar to class diagrams but model the system's state rather than its static structure.
3. Data Flow Diagram	<ul style="list-style-type: none"> Although typically used to show data movement, DFDs can also be considered a form of structural modeling as they define how different components (e.g., processes, stores,

	and external entities) are organized and interact within the system.
4. Component Diagram	<ul style="list-style-type: none"> Represents how a system is divided into components and how these components interact with each other. It shows how the physical software components (such as libraries, executables, or databases) are organized.
5. Deployment Diagram	<ul style="list-style-type: none"> Depicts the hardware used in system deployment and how software components are allocated to this hardware.

1. Class diagram

- Describes the structure of a system by showing the system's classes, their attributes, operations (or methods) and their relationships between the classes.
- The upper section encompasses class name
- Middle section constitutes the attributes.
- Lower section contains methods or operations
- Relationships in Class diagram
- Relationships are:



Relationships:

1. **Dependency** : Eg: Student_Name is dependent on Student_Id
2. **Association** : describes connection between two or more objects
3. **Generalization** : Relationship between parent class and child class
4. **Aggregation** : subset of association. Eg. Company encompasses a number of employees
5. **Composition**: It represents a whole-part relationship.
6. **Multiplicity**: Multiple patients are admitted to one hospital

Sample Class Diagram

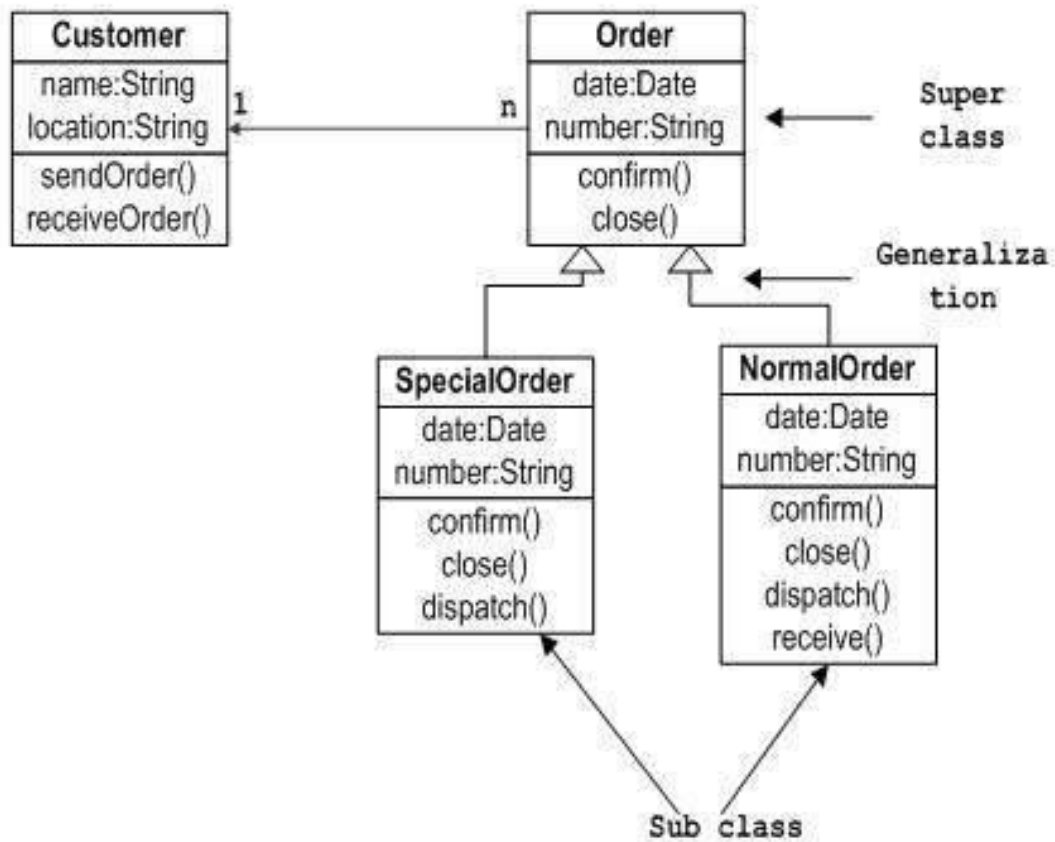


Figure: A sample data for Order Management

2. Object Diagram

Focuses on instances of classes (objects) and their relationships at a specific point in time. Object diagrams are similar to class diagrams but model the system's state rather than its static structure.

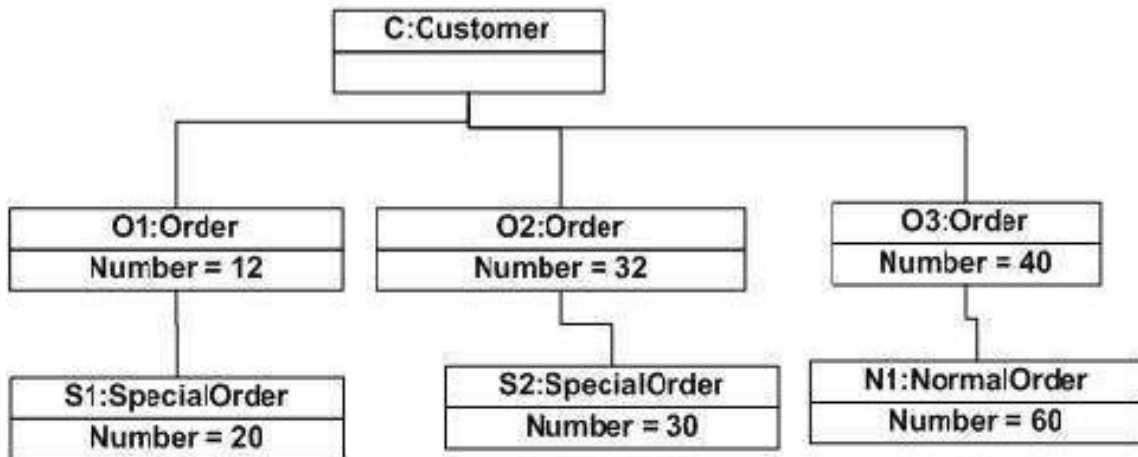

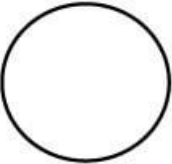
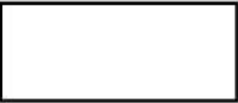
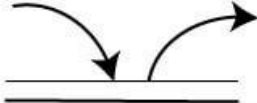


Figure: Object Diagram for an Order Management System

1. Data Flow Diagram

Although typically used to show data movement, DFDs can also be considered a form of structural modeling as they define how different components (e.g., processes, stores, and external entities) are organized and interact within the system.

Symbol	Name	Function
	Data flow	Used to Connect Processes to each other, to sources or Sinks; the arrow head indicates direction of data flow.
	Process	Performs Some transformation of Input data to yield output data.
	Source of Sink (External Entity)	A Source of System inputs or Sink of System outputs.
	Data Store	A repository of data; the arrow heads indicate net inputs and net outputs to store.

Symbols for Data Flow Diagrams

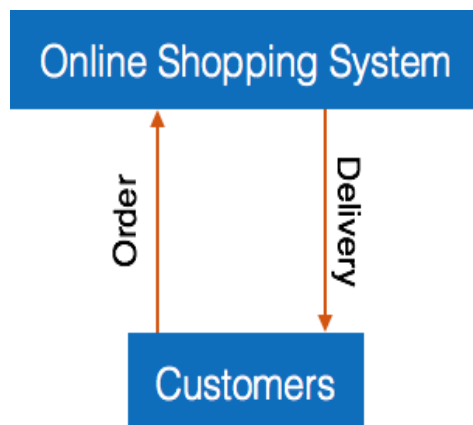


Figure 1: Level 0 DFD for Online Shopping System

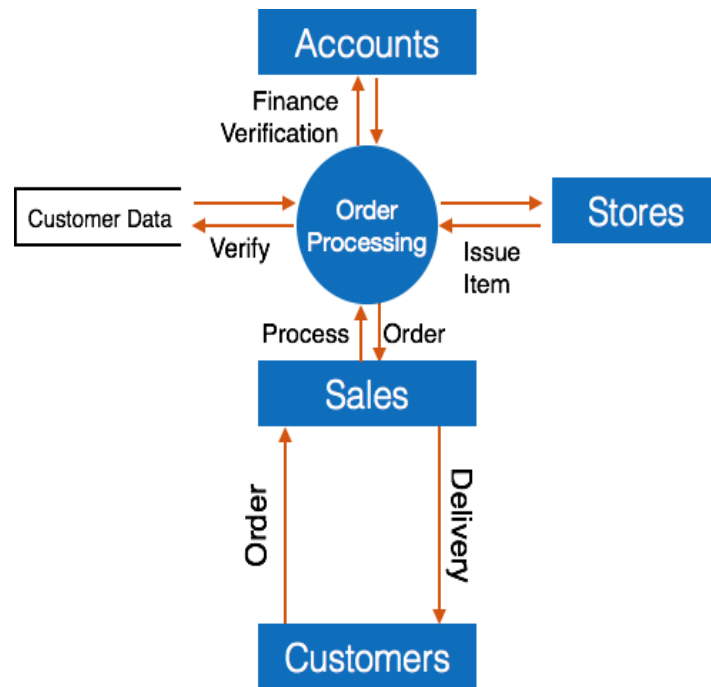
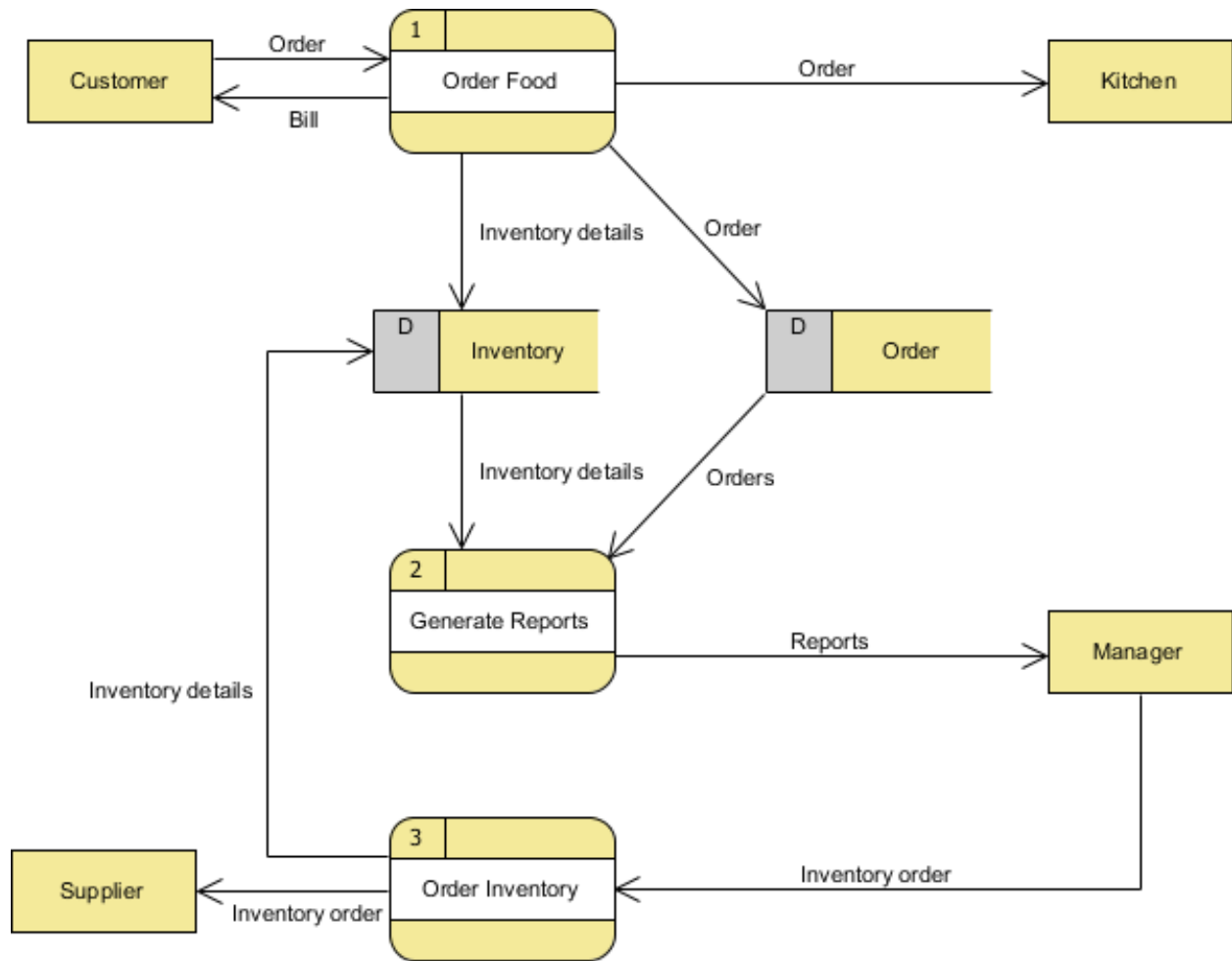


Figure 2: level 1 DFD for Online Shopping Systems.



2. Component Diagram

- Represents how a system is divided into components and how these components interact with each other.
- It shows how the physical software components (such as libraries, executables, or databases) are organized.

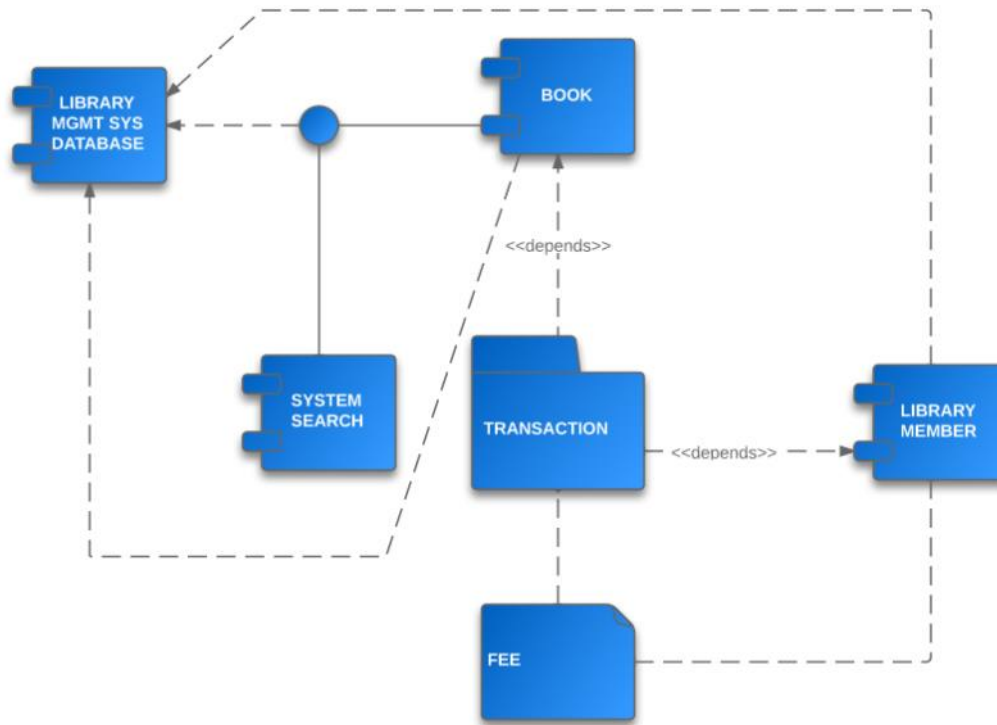


Figure: Component Diagram for Library management.

3. Deployment Diagram

Depicts the hardware used in system deployment and how software components are allocated to this hardware.

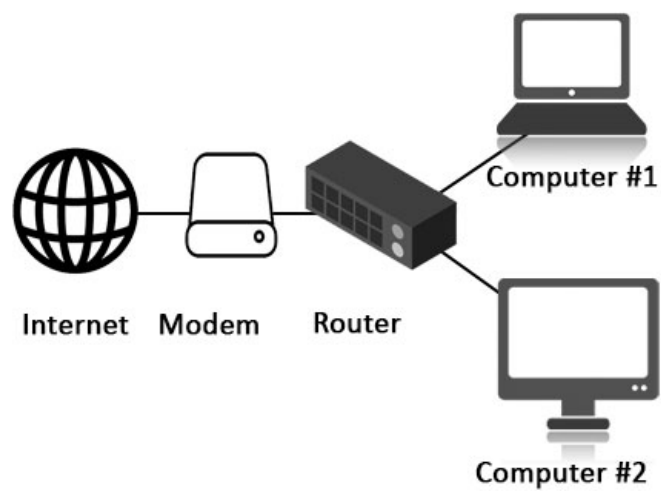
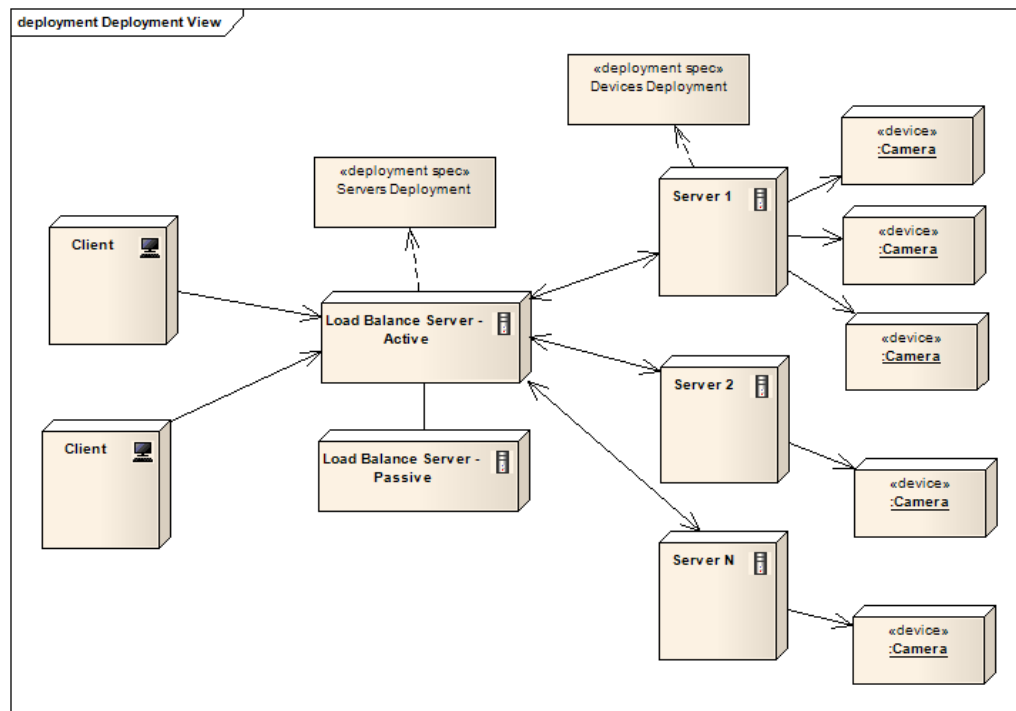


Figure: Deployment diagram.

3.4 Requirements Engineering Process

Requirement Engineering (RE) is the process of defining, documenting, and maintaining the requirements for a software system. It involves understanding what the stakeholders need from the software, ensuring that these needs are captured accurately, and managing changes to requirements throughout the development lifecycle.

Importance of Requirement Engineering:

- **Foundation for Success:** Accurate and comprehensive requirements are critical for the success of a software project.
- **Cost Efficiency:** Early identification of requirements can reduce costly changes later in the development process.
- **Improved Stakeholder Satisfaction:** Involving stakeholders throughout the RE process helps ensure that the final product meets their needs and expectations.
- **Enhanced Quality:** A clear understanding of requirements leads to better design, implementation, and testing, resulting in higher-quality software.

Phases of the Requirement Engineering Process

1. Requirements Elicitation

- **Objective:** Gather requirements from stakeholders, including customers, users, and other interested parties.
- **Techniques:**
 - **Interviews:** Conduct discussions with stakeholders to gather their needs and expectations.
 - **Surveys/Questionnaires:** Distribute structured forms to collect information from a larger audience.
 - **Workshops:** Facilitate group discussions to gather insights and clarify requirements.
 - **Observation:** Analyze how users interact with existing systems to identify needs.
 - **Prototyping:** Create mock-ups of the system to help stakeholders visualize and refine their requirements.

2. Requirements Analysis

- **Objective:** Organize and prioritize the gathered requirements to identify conflicts, ambiguities, and dependencies.

- **Activities:**
 - **Categorization:** Group requirements into functional and non-functional categories.
 - **Feasibility Analysis:** Assess whether the requirements can be met within budget, time, and technology constraints.
 - **Prioritization:** Determine the importance of each requirement and its impact on system functionality.
 - **Conflict Resolution:** Identifying and addressing conflicting stakeholder need.

3. Requirements Specification

- **Objective:** Document the requirements clearly and unambiguously for stakeholders and the development team.
- **Deliverables:**
 - **Requirements Specification Document:** A comprehensive document detailing all gathered and analyzed requirements.
 - **Use Cases/User Stories:** Detailed descriptions of how users will interact with the system, outlining specific scenarios and expected outcomes.
 - **Modeling:** Use visual models (such as UML diagrams) to represent requirements and their relationships.

4. Requirements Validation

- **Objective:** Ensure that the documented requirements meet the stakeholders' needs and are feasible for implementation.
- **Activities:**
 - **Review Meetings:** Conduct discussions with stakeholders to review and validate requirements.
 - **Traceability:** Establish links between requirements and their sources to track changes and ensure coverage.
 - **Test Case Development:** Create initial test cases based on the requirements to ensure that they are testable.

5. Requirements Management

- **Objective:** Monitor and control changes to requirements throughout the software development lifecycle.
- **Activities:**

- **Change Management:** Implement processes to handle requests for changes to requirements, including assessment and approval.
- **Version Control:** Maintain versions of requirement documents to track changes over time.
- **Communication:** Keep stakeholders informed about changes and updates to requirements.

3.4.1 Feasibility Study

A **feasibility study** is an assessment of the practicality and viability of a proposed project or system. In software engineering, it evaluates whether the project is achievable, cost-effective, and likely to succeed within the given constraints, such as time, resources, and technology.

Purpose of a Feasibility Study:

- **Assess Viability:** Determine if the project can be completed successfully.
- **Identify Risks:** Highlight potential challenges and risks associated with the project.
- **Inform Decision-Making:** Provide stakeholders with the necessary information to decide whether to proceed with the project.
- **Resource Allocation:** Help in estimating resources, costs, and time required for the project.

Benefits of a Feasibility Study:

- **Informed Decisions:** Provides a solid basis for stakeholders to decide whether to proceed with the project.
- **Risk Mitigation:** Identifies potential risks early, allowing for better planning and management.
- **Resource Optimization:** Ensures that resources are allocated effectively and efficiently.
- **Increased Success Rates:** Projects that undergo feasibility studies tend to have higher success rates due to careful planning and analysis.

Types of Feasibility Studies:

1. Technical Feasibility

- **Objective:** Assess whether the proposed technology and architecture can support the project.

- **Considerations:**
 - Availability of hardware and software resources.
 - Compatibility with existing systems.
 - Technical skills and expertise required.
- **Outcome:** Determines if the technical approach is feasible.

2. Economic Feasibility

- **Objective:** Evaluate the cost-effectiveness of the project.
- **Considerations:**
 - Cost-Benefit Analysis: Compare projected costs against expected benefits.
 - Return on Investment (ROI): Assess the potential return on the investment made in the project.
 - Budget Constraints: Ensure the project aligns with the organization's financial capacity.
- **Outcome:** Determines if the project is financially viable.

3. Operational Feasibility

- **Objective:** Examine whether the organization can support the project operationally.
- **Considerations:**
 - Impact on existing operations and processes.
 - User acceptance and readiness for change.
 - Training and support requirements for users.
- **Outcome:** Determines if the project is acceptable to stakeholders and fits into current operations.

4. Legal Feasibility

- **Objective:** Identify any legal issues that may affect the project.
- **Considerations:**
 - Compliance with laws and regulations.
 - Licensing issues related to software and technology.
 - Intellectual property concerns.
- **Outcome:** Determines if the project adheres to legal standards.

5. Schedule Feasibility

- **Objective:** Assess whether the project can be completed within a specified timeframe.
- **Considerations:**

- Project timeline and milestones.
- Resource availability and constraints.
- Potential delays and their impact.
- **Outcome:** Determines if the project schedule is realistic.

3.4.2 Requirements Elicitations, Analysis, Modeling, Specifications

a) Requirements Elicitations:

Requirement elicitation is the process of gathering and understanding the needs and expectations of stakeholders for a software project. It is a crucial step in the requirement engineering process that involves interactions between stakeholders and the development team to identify, clarify, and document requirements.

Importance of Requirement Elicitation:

- **Foundation for Development:** Accurate requirements are essential for designing and building effective software systems.
- **Stakeholder Satisfaction:** Involving stakeholders ensures their needs are met, increasing satisfaction with the final product.
- **Reduced Risk of Miscommunication:** Clear communication helps avoid misunderstandings and misinterpretations.
- **Cost Savings:** Early identification of requirements can prevent costly changes and rework later in the development process.

Stakeholders Involved in Requirement Elicitation:

- **End Users:**
- **Customers:** Clients or organizations that commission the project and define their expectations.
- **Subject Matter Experts (SMEs):** Individuals with specialized knowledge relevant to the project domain.
- **Developers:** Technical team members who will design and implement the system.

- **Project Managers:** Individuals responsible for overseeing the project and ensuring that requirements align with project goals.

Techniques for Requirement Elicitation:

1. Interviews

- **Description:** Conducting one-on-one or group discussions with stakeholders to gather detailed information about their needs and preferences.
- **Advantages:** Provides in-depth insights; allows for follow-up questions and clarification.

2. Surveys and Questionnaires

- **Description:** Distributing structured forms to gather information from a larger audience.
- **Advantages:** Efficient for collecting data from many stakeholders; easy to analyze quantitative responses.

3. Workshops

- **Description:** Facilitated group discussions involving multiple stakeholders to gather insights and reach consensus on requirements.
- **Advantages:** Encourages collaboration; helps identify conflicting requirements early.

4. Observation

- **Description:** Analyzing how users interact with existing systems to identify needs and workflows.
- **Advantages:** Provides real-world context; highlights user pain points that may not be verbally expressed.

5. Prototyping

- **Description:** Creating mock-ups or models of the system to help stakeholders visualize and refine their requirements.
- **Advantages:** Facilitates feedback; helps stakeholders articulate their needs more clearly.

6. Use Cases and User Stories

- **Description:** Documenting scenarios of how users will interact with the system to clarify functional requirements.

- **Advantages:** Provides context for requirements; emphasizes user goals and tasks.

7. Brainstorming

- **Description:** Generating a list of ideas and requirements through collaborative discussions.
- **Advantages:** Encourages creative thinking; can lead to innovative solutions.

8. Document Analysis

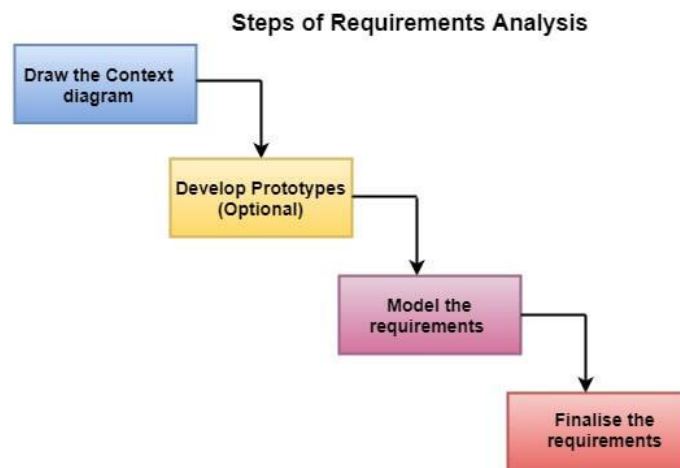
- **Description:** Reviewing existing documentation (e.g., manuals, specifications) related to the project to gather relevant information.
- **Advantages:** Provides historical context; identifies existing requirements and constraints.

b) Requirements Analysis:

Requirements Analysis is the process of defining the expectations of the users for an application that is to be built or modified. It involves all the tasks that are conducted to identify the needs of different stakeholders. Therefore, requirements analysis means to analyze, document, validate and manage software or system requirements.

Requirement analysis is significant and essential activity after elicitation. We analyze, refine, and scrutinize the gathered requirements to make consistent and unambiguous requirements. This activity reviews all requirements and may provide a graphical view of the entire system. After the completion of the analysis, it is expected that the understandability of the project may improve significantly. Here, we may also use the interaction with the customer to clarify points of confusion and to understand which requirements are more important than others.

The various steps of requirement analysis are shown in figure below:



- (i) **Draw the context diagram:** The context diagram is a simple model that defines the boundaries and interfaces of the proposed systems with the external world. It identifies the entities outside the proposed system that interact with the system.
- (ii) **Development of a Prototype (optional):** One effective way to find out what the customer wants is to construct a prototype, something that looks and preferably acts as part of the system they say they want.
- (iii) **Model the requirements:** This process usually consists of various graphical representations of the functions, data entities, external entities, and the relationships between them. The graphical view may help to find incorrect, inconsistent, missing, and superfluous requirements. Such models include the Data Flow diagram, Entity-Relationship diagram, Data Dictionaries, State-transition diagrams, etc.
- (iv) **Finalize the requirements:** After modeling the requirements, we will have a better understanding of the system behavior. The inconsistencies and ambiguities have been identified and corrected. The flow of data amongst various modules has been analyzed. Elicitation and analyze activities have provided better insight into the system. Now we finalize the analyzed requirements, and the next step is to document these requirements in a prescribed format.

c) Requirements Modeling:

Requirement modeling is the process of creating abstract representations of the software system's requirements to understand, visualize, and communicate them effectively. These models serve as a bridge between the stakeholders' needs and the software design, helping ensure that the system meets its intended purpose.

Types of Requirement Models

1. Context Models

- **Description:** Illustrate the system's environment, showing the boundaries of the system and its interactions with external entities (users, systems, etc.).
- **Purpose:** Define what is inside and outside the system, helping identify stakeholders and external interfaces.

2. Behavioral Models

- **Description:** Capture the dynamic aspects of the system, showing how it responds to various inputs and conditions over time.

- **Examples:** Use case diagrams, sequence diagrams, state diagrams.
- **Purpose:** Help stakeholders understand how the system will behave in different scenarios.

3. Structural Models

- **Description:** Represent the static aspects of the system, detailing its components and their relationships.
- **Examples:** Class diagrams, entity-relationship diagrams.
- **Purpose:** Provide a clear view of the system's architecture and data organization.

4. Data Models

- **Description:** Define how data is structured, stored, and manipulated within the system.
- **Examples:** Relational models, object-oriented models.
- **Purpose:** Ensure that data requirements are well-understood and aligned with the system's needs.

5. Functional Models

- **Description:** Focus on the functions the system must perform, outlining the system's capabilities.
- **Examples:** Functional decomposition diagrams, data flow diagrams.
- **Purpose:** Identify the functional requirements and their interactions.

d) Requirements Specifications:

Requirement specification is the process of formally documenting the requirements of a software system in a clear, detailed, and unambiguous manner. This document serves as a reference point for all stakeholders, guiding the design, development, and testing phases of the software lifecycle. By documenting requirements clearly and comprehensively, development teams can ensure that they build software systems that meet user needs and expectations.

Purpose:

- Ensure clarity and precision in understanding system expectations.
- Serve as a foundational guide for development.
- Provide criteria for validation and verification.
- Facilitate communication among stakeholders.

Components:

1. **Introduction:** Overview of the system's purpose and scope.
2. **General Description:** High-level context and stakeholder information.
3. **Specific Requirements:**
 - **Functional Requirements:** Detailed system functionalities and behaviors.
 - **Non-Functional Requirements:** Qualities like performance, security, and usability.
4. **Use Cases/User Stories:** Scenarios for user interactions with the system.
5. **Constraints and Assumptions:** Limitations to consider during development.
6. **Acceptance Criteria:** Conditions for stakeholder acceptance of the system.
7. **Traceability Matrix:** Mapping of requirements to design and testing artifacts.

Types:

- **Business Requirements Specification (BRS):** High-level business needs.
- **User Requirements Specification (URS):** Detailed user expectations.
- **System Requirements Specification (SRS):** Comprehensive documentation of all system requirements.

Techniques:

- Use of clear natural language.
- Modeling techniques (e.g., UML diagrams).
- Prototyping for clarification.
- Checklists for completeness and clarity.

Importance:

- Reduces ambiguity and misunderstandings.
- Improves project success rates.
- Enhances traceability throughout development.
- Facilitates effective change management.

3.4.3 Requirements Validation

Requirements validation is the process of ensuring that the documented requirements for a software system accurately reflect the needs and expectations of stakeholders. It involves checking that the requirements are complete, feasible, realistic, and consistent, and that they will result in a system that meets the intended goals.

Purpose:

- Ensure completeness of requirements.
- Identify and resolve conflicts or inconsistencies.
- Assess the feasibility of implementing requirements.
- Validate alignment with stakeholder expectations.
- Mitigate risks by identifying potential issues early.

Key Activities:

1. **Review and Inspection:** Formal reviews involving stakeholders to identify errors or omissions.
2. **Prototyping:** Creating mock-ups to visualize and gather feedback on requirements.
3. **Use Case Analysis:** Developing use cases to illustrate user interactions with the system.
4. **Walkthroughs:** Structured group discussions to review requirements collaboratively.
5. **Modeling:** Using visual representations (like UML diagrams) to identify relationships among requirements.
6. **Testing Requirements:** Establishing acceptance criteria for each requirement to guide verification.

Validation Techniques:

- **Traceability Matrix:** Linking requirements to design and test cases to ensure all are addressed.
- **Stakeholder Interviews:** Engaging stakeholders to clarify needs and confirm documented requirements.
- **Requirements Workshops:** Collaborative sessions with stakeholders for collective review.
- **Checklists:** Using standard checklists to verify completeness and correctness of requirements.

3.5 Object Oriented Analysis

Object-Oriented Analysis (OOA) is a method of analyzing a problem domain by identifying and modeling the objects that will be used to solve that problem. It focuses on understanding the requirements of a system through the lens of objects, which encapsulate both data and behavior.

Key Concepts of Object-Oriented Analysis:

1. **Objects:** Fundamental building blocks in OOA, representing real-world entities or concepts that have attributes (data) and behaviors (methods). For example, in a library management system, an object could be a "Book" with attributes like title and author and methods like checkout and return.
2. **Classes:** A blueprint for creating objects. It defines the attributes and behaviors that the objects of that class will have. For example, a "Book" class defines what properties every book object will have.
3. **Inheritance:** A mechanism that allows one class (subclass) to inherit the attributes and methods of another class (superclass). This promotes code reuse and establishes a hierarchical relationship. For instance, a "Magazine" class could inherit from the "Publication" class.
4. **Encapsulation:** The bundling of data and methods that operate on that data within a single unit (class), restricting access to some components. This principle protects the internal state of an object from direct access and modification.
5. **Polymorphism:** The ability of different classes to respond to the same method call in different ways. This allows for flexibility and the ability to define multiple behaviors for the same operation based on the object invoking it.

Process of Object-Oriented Analysis:

1. **Problem Definition:** Identify the problem to be solved and the objectives of the system.
2. **Identify Objects:** Determine the objects that are relevant to the problem domain. This can be achieved through:
 - **Use Case Analysis:** Identifying the interactions between users and the system.
 - **Domain Modeling:** Creating a conceptual model that captures the essential entities and their relationships.
3. **Define Object Responsibilities:** For each identified object, specify what data it holds and what operations it can perform.

4. **Identify Relationships:** Determine how the objects interact with each other. This can include associations, aggregations, and dependencies between objects.
5. **Develop Use Cases:** Create use case diagrams to illustrate the interactions between users (actors) and the system, outlining the different ways users will interact with the objects.
6. **Refine Models:** Continuously refine and adjust the models based on feedback and additional insights, ensuring that they accurately represent the system's requirements.

End of chapter