What are Functional Requirements?

These are the requirements that the end user specifically demands as basic facilities that the system should offer. All these functionalities need to be necessarily incorporated into the system as a part of the contract.

- These are represented or stated in the form of input to be given to the system, the operation performed and the output expected.
- They are the requirements stated by the user which one can see directly in the final product, unlike the non-functional requirements.

Examples:

- What are the features that we need to design for this system?
- What are the edge cases we need to consider, if any, in our design?

What are Non-Functional Requirements?

These are the quality constraints that the system must satisfy according to the project contract. The priority or extent to which these factors are implemented varies from one project to another. They are also called non-behavioral requirements. They deal with issues like:

- Portability
- Security
- Maintainability
- Reliability
- Scalability
- Performance

- Reusability
- Flexibility

Examples:

- Each request should be processed with the minimum latency?
- System should be highly available.

What are Extended Requirements?

These are basically "nice to have" requirements that might be out of the scope of the System.

Example:

- Our system should record metrices and analytics.
- Service heath and performance monitoring.

Examples of Functional and Non-functional Requirements

Let's consider a couple of examples to illustrate both types of requirements:

1. Online Banking System

1. Functional Requirements:

- Users should be able to log in with their username and password.
- Users should be able to check their account balance.
- Users should receive notifications after making a transaction.

2. Non-functional Requirements:

- The system should respond to user actions in less than 2 seconds.
- All transactions must be encrypted and comply with industry security standards.
- The system should be able to handle 100 million users with minimal downtime.

2. Food Delivery App

1. Functional Requirements

- Users can browse the menu and place an order.
- Users can make payments and track their orders in real time.

2. Non-functional Requirements:

- The app should load the restaurant menu in under 1 second.
- The system should support up to 50,000 concurrent orders during peak hours.
- The app should be easy to use for first-time users, with an intuitive interface.

Differences between Functional Requirements and Non-Functional Requirements:

Below are the differences between Functional Requirements and Non-Functional Requirements:

Aspect	Functional Requirements	Non-Functional Requirements
Definition	Describes what the system should do, i.e., specific functionality or tasks.	Describes how the system should perform, i.e., system attributes or quality.
Purpose	Focuses on the behavior and features of the system.	Focuses on the performance, usability, and other quality attributes.
Scope	Defines the actions and operations of the system.	Defines constraints or conditions under which the system must operate.

Examples	User authentication, data input/output, transaction processing.	Scalability, security, response time, reliability, maintainability.
Measurement	Easy to measure in terms of outputs or results.	More difficult to measure, often assessed using benchmarks or SLAs.
Impact on Development	Drives the core design and functionality of the system.	Affects the architecture and overall performance of the system.
Focus on User Needs	Directly related to user and business requirements.	Focuses on user experience and system performance.

Documentation	Typically documented in use cases, functional specifications, etc.	Documented through performance criteria, technical specifications, etc.
Evaluation	Can be tested through functional testing (e.g., unit or integration tests).	Evaluated through performance testing, security testing, and usability testing.
Dependency	Determines what the system must do to meet user needs.	Depends on how well the system performs the required tasks.

How to Gather Functional and Non-functional Requirements

Gathering requirements involves multiple approaches and collaboration between the development team, stakeholders, and end-users:

1. Functional Requirements:

- **Interviews:** Talk to stakeholders or users to understand their needs.
- **Surveys:** Distribute questionnaires to gather input from a larger audience.
- **Workshops:** Host sessions to brainstorm features and gather feedback.

2. Non-functional Requirements:

- **Performance Benchmarks:** Consult with IT teams to set expectations for performance and load.
- **Security Standards:** Consult with security experts to define the best practices for data protection.
- **Usability Testing:** Test the system to find areas where users might struggle and refine the interface.

Importance of Balancing Both Functional and Non-Functional Requirements

As functional and non-functional requirements address distinct but equally significant components of the system, it is essential that they be balanced in system design. Here's why balancing them is essential:

- **Improves User Experience**: A system that functions but is slow, unresponsive, or challenging to operate could be the consequence of concentrating only on the functional requirements. Usability,

performance, and availability are examples of non-functional criteria that have a big influence on how users interact with the system and how satisfied they are.

- **Enhances System Performance**: Scalability, dependability, and security are examples of non-functional features that guarantee the system operates effectively in real-world situations. Failure may ensue from a system that satisfies functional requirements but is unable to scale or manage a high user volume. By keeping these in balance, the system is guaranteed to function properly.
- **Prevents Bottlenecks and Failures**: Systems that function well but lack security or dependability are more likely to experience malfunctions, breaches, or outages. The system is made more robust and less vulnerable to bottlenecks by taking non-functional criteria into account.
- **Reduces Long-Term Costs**: Systems designed with only functional requirements in mind often encounter scalability and maintainability issues as they grow. Addressing non-functional requirements early on prevents costly changes, re-architecting, or performance enhancements later.
- **Supports System Evolution**: Non-functional requirements such as maintainability and extensibility make future updates, feature additions, and system scaling easier and less disruptive. This

balance ensures that the system can evolve and adapt over time without significant hurdles.

Common Challenges in Defining these Requirements

Defining both functional and non-functional requirements in system design can be challenging due to several factors. Here are some common challenges:

- **Ambiguity in Requirements:** Requirements are often unclear, making it difficult to interpret what the system should do (functional) and how it should perform (non-functional).
- **Changing Requirements:** As projects evolve, functional and non-functional requirements often change due to shifting business goals, market trends, or user needs.
- **Difficulty in Prioritization:** Deciding which functional and non-functional requirements are most important is often difficult. Functional needs can overshadow non-functional ones, such as security or scalability.
- **Measuring Non-Functional Requirements:** Functional requirements are often straightforward to test, but non-functional requirements like usability, scalability, and reliability are harder to measure and validate.

- **Overlapping or Conflicting Requirements:** Sometimes, functional and non-functional requirements can overlap or conflict. For example, adding complex security features (non-functional) might slow down the performance (another non-functional requirement).

User Requirements

User requirements define what users need from an application or software to solve a problem or achieve an objective. These requirements are typically expressed in natural language and focus on the user's perspective, outlining the desired functionality and features of the software.

Key Characteristics of User Requirements:

User-centric:

User requirements focus on the needs and expectations of the people who will be using the software.

High-level:

They are generally described at a high level, avoiding technical details.

Clear and Unambiguous:

Requirements should be easy to understand and interpret, avoiding confusion.

Specific, Measurable, Achievable, Relevant, and Time-bound (SMART):

Good user requirements are specific, measurable, achievable, relevant, and time-bound, meaning they are clear, can be tested, realistic, aligned with business goals, and have deadlines.

Examples:

Functionality: A user might need to be able to search for items by name, filter results by category, and sort them by price.

Data: The software should store user profiles, product information, and transaction details.

Security: Users should be able to log in securely and have their data protected.

Performance: Search results should be returned within a reasonable time.

Distinction between User Requirements and System Requirements:

User Requirements: Focus on what the user needs from the system.

System Requirements: Focus on the technical specifications of the system itself, such as hardware, software, and infrastructure needed to run the application.

User Requirements Specification (URS):

A URS document outlines the business needs and what users require from the system.

It is typically written early in the validation process, before the system is created.

The URS is not intended to be a technical document; it should be understandable by those with a general knowledge of the system.

System Requirements

System requirements refer to the specifications needed for a software application,game, or any digital tool to operate efficiently on a computer system. These requirements can encompass a broad range of components, including hardware, software, and connectivity aspects. Meeting these requirements ensures that the software runs smoothly without performance issues or compatibility problems.

System requirements are typically divided into two categories: minimum and recommended. The ***minimum requirements*** indicate the least capable hardware and software setup on which the software can run. This setup might allow the software to function but not necessarily at optimal performance levels. The ***recommended requirements*** provide a specification that ensures a smooth and optimal user experience, offering better performance, stability, and overall functionality.

Importance of System Requirements

Properly defined system requirements are crucial for several reasons:

- Optimal Performance - Ensuring that the system meets the necessary requirements helps in achieving the best possible performance for the software. For example, a graphics-intensive game like "PhotoMaster Pro" requires a powerful GPU and sufficient RAM to render high-quality images and effects smoothly.
- User Experience - Adequate system requirements contribute to a seamless and enjoyable user experience, reducing frustration due to lagging or crashing applications. If users encounter frequent performance issues, they are likely to become frustrated and dissatisfied with the software.
- Compatibility - System requirements ensure that the software is compatible with the user's hardware and operating system, preventing installation and execution issues. Compatibility checks help in avoiding scenarios where the software might not launch or function incorrectly due to incompatible environments.
- Resource Planning - For businesses and IT departments, understanding system requirements aids in planning and allocating resources effectively, avoiding unnecessary expenses on incompatible or inadequate hardware. This is particularly important for large-scale deployments where the cost of hardware upgrades can be substantial.

Types of System Requirements

System requirements can be broadly categorized into three main types: hardware requirements, software requirements, and connectivity requirements. Each of these categories addresses different aspects of the system's configuration necessary to support the software.

Hardware Requirements

Hardware requirements define the physical components needed to run the software. These typically include:

- Architecture - The type of processor architecture (e.g., x86, x64) that the software supports. Different software might require specific architectures to leverage certain features or optimizations.
- Processing Power - The minimum and recommended CPU specifications, such as the number of cores and clock speed. High-performance applications, such as video editing tools, require powerful processors to handle complex computations efficiently.
- Memory - The amount of RAM required for the software to run efficiently, with specifications for both minimum and recommended memory. More RAM allows the software to handle larger datasets and multitask more effectively.
- Storage - The amount of disk space needed for installation and operation, often including both minimum and recommended storage capacities. Storage requirements also consider the type of storage, such as SSDs for faster read/write speeds compared to traditional HDDs.
- Display Adapter - Specifications for the graphics card, including GPU model, VRAM, and supported features like DirectX or OpenGL versions. For gaming and graphic design software, a robust GPU is essential to render graphics smoothly.

- Peripherals - Additional hardware components required, such as keyboards, mice, or VR headsets. Specialized peripherals might be necessary for certain types of software, like gaming controllers or drawing tablets for digital art applications.

Software Requirements

Software requirements define the necessary software environment for the application to function properly. These include:

- Operating System - The compatible operating systems (e.g., Windows, macOS, Linux) and specific versions required. Different operating systems have different capabilities and limitations, affecting software performance and compatibility.
- APIs - Required application programming interfaces (APIs) such as DirectX, Vulkan, or OpenGL. APIs enable software to interface with hardware and other software components effectively.
- Drivers - Necessary drivers for hardware components, particularly for graphics cards and other peripherals. Updated drivers ensure that hardware components function correctly and efficiently with the software.
- Web Browser - Specific web browser versions needed if the application relies on web-based technologies or components. Web-based applications might require modern browsers that support the latest web standards.
- Runtime Environments - Any additional runtime environments or frameworks required, such as Java, .NET, or Python. These environments provide the necessary runtime support for executing software applications.

Connectivity Requirements

Connectivity requirements specify the network-related conditions necessary for the software to operate, particularly for online applications or games. These include:

- Connection Type - The type of network connection needed, such as Wi-Fi, wired Ethernet, or wireless. The connection type can affect the stability and speed of data transmission.
- Speed - Minimum and recommended internet connection speeds for optimal performance, particularly for online multiplayer games or cloud-based applications. Higher speeds ensure low latency and faster data transfers.
- Protocols Supported - Network protocols that the software must support, such as TCP/IP, HTTP/HTTPS, or FTP. These protocols facilitate communication and data exchange over the network.

Interface Requirements

Interface specification, in the context of software development and computer systems, refers to a detailed description or set of rules that define how different software components or modules should interact with each other.

It acts as a contract or agreement that ensures seamless communication and integration between various parts of a software system.

In simpler terms, an interface specification outlines the rules and guidelines for how different parts of a software application should “talk” to each other, exchange data, and cooperate to perform specific tasks. These interfaces can be between software modules, software and hardware components, or even between different software systems.

Types of interface specification-

There are three types of Interface specification:

1) Procedural interfaces.
2) Data structures.
3) Representations of data

Procedural interface-

Procedural interfaces where existing programs or sub-systems offer a range of services that are accessed by calling interface procedures. In simple words it is used for calling the existing programs by the new programs These interfaces are sometimes called Application Programming Interfaces (APLs).

Data Structure-

Data structures have been passed from one sub-system to another. Graphical data models are the best notations for this type of description.

Representation of data-

Representations of data (such as the ordering of bits) have been established for an existing sub-system. These interfaces are most common in embedded, real-time systems. Some programming languages such as Ada (although not Java) support this level of Specifications

Key Points about Interface Specification

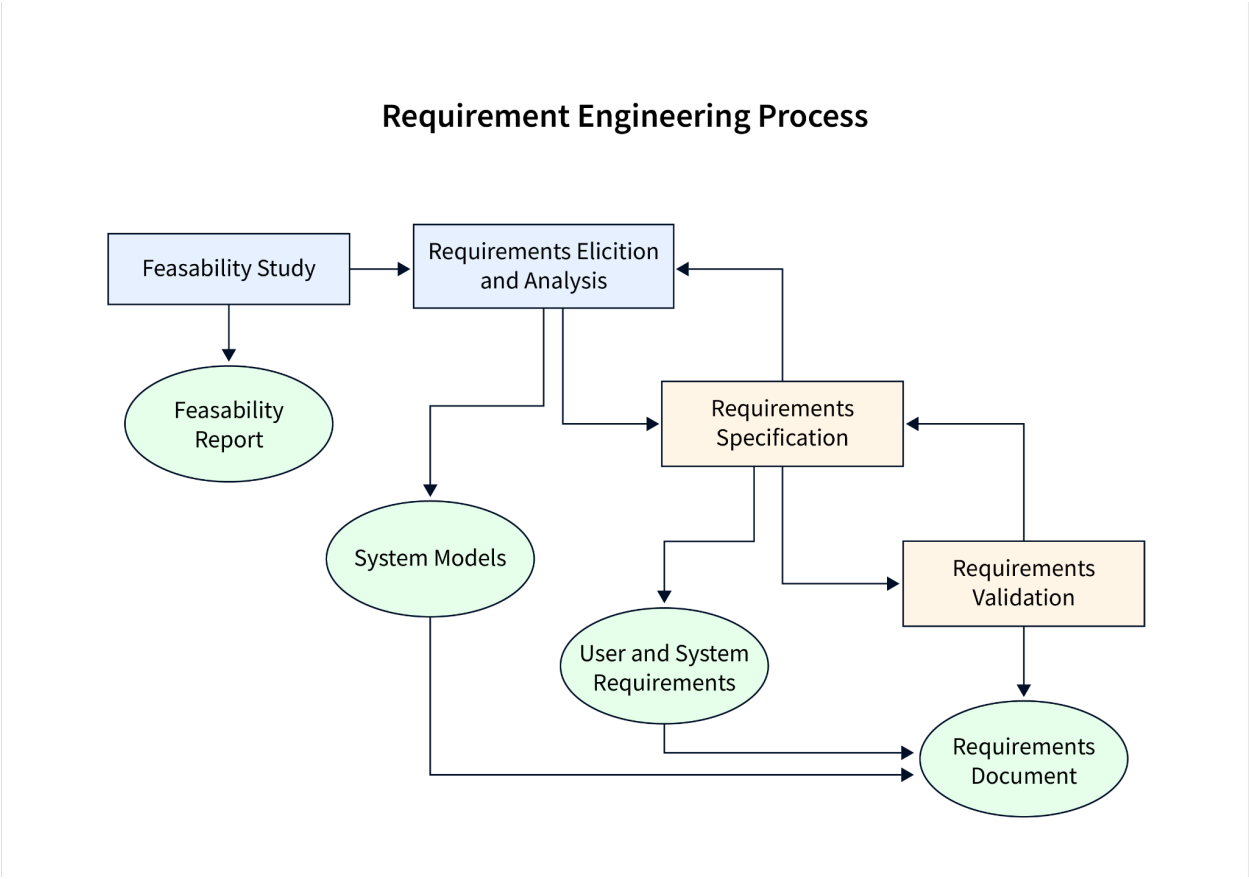
Standardization : Interface specifications standardize the communication between components, making it easier for developers to understand how to interact with other parts of the system.

Abstraction : Interfaces hide the implementer on details of a component, allowing other parts of the system to interact with it without needing to know the internal complexities.

Flexibility : By defining clear interfaces, components can be easily replaced or upgraded without affecting other parts of the system, as long as they adhere to the same interface specification.

Requirement Engineering Process

Requirement Engineering is a fundamental process in software development that focuses on understanding, documenting, validating, and managing the needs and constraints of stakeholders to ensure the successful development of a software system. The process typically consists of several steps to ensure that the final software product meets the needs and expectations of the users and stakeholders.



1.Feasibility Study: This is the initial phase of the requirement engineering process. It involves evaluating the feasibility of the proposed software

project. The main objective is to determine whether the project is technically, economically, and operationally viable. During this phase, the following aspects are considered:

- Technical Feasibility: Can the proposed system be developed with the available technology and resources?
- Economic Feasibility: Is the project economically viable in terms of cost and benefits?
- Operational Feasibility: Will the system work effectively within the existing organizational structure and processes?

2. Requirement Elicitation and Analysis: This phase involves gathering and analyzing requirements from stakeholders, including users, customers, and domain experts. The goal is to understand the needs of the system and translate them into specific software requirements. This process involves several techniques, such as:

- Interviews: Direct communication with stakeholders to gather information.
- Surveys: Distributing questionnaires to collect opinions and preferences.
- Workshops: Group sessions to brainstorm and gather requirements collaboratively.

- Observation: Actively observing users to understand their workflow and needs.
- Prototyping: Building a preliminary version of the software to elicit feedback.

  Once requirements are gathered, they are analyzed for consistency, completeness, and feasibility. Conflicts and ambiguities are resolved, and the requirements are organized into different categories based on their characteristics (e.g., functional, non-functional).

3. Software Requirement Specification: In this phase, the gathered and analyzed requirements are documented formally. The Software Requirement Specification (SRS) document is created, which serves as a blueprint for the development process. The SRS includes the following components:

- Functional Requirements: Descriptions of what the system should do in terms of specific functions and features.
- Non-functional Requirements: Constraints and qualities the system must possess (e.g., performance, security).
- User Interfaces: Descriptions of how users will interact with the system.

- Data Requirements: Details about the data the system will store, process, and manage.

4. Software Requirement Validation: This step involves validating the documented requirements to ensure they accurately represent the stakeholders' needs and expectations. The aim is to identify and rectify any errors or misunderstandings before development begins. Techniques used for requirement validation include:

- Reviews: Expert reviews and walkthroughs to identify issues in the SRS.
- Prototyping: Building a working model of the software to validate requirements.
- Simulation: Simulating the software's behavior to test its alignment with requirements.

5. Software Requirement Management: Requirement management is an ongoing process that involves maintaining and tracking changes to requirements throughout the software development lifecycle. It includes activities such as:

- Version Control: Managing different versions of the requirements document.

- Change Control: Managing and documenting changes to requirements and their impact.
- Traceability: Establishing links between requirements and design, implementation, and testing.
- Prioritization: Assigning priorities to requirements to guide development efforts.
- Communication: Ensuring stakeholders are informed about changes and updates to requirements.

System Modeling

System modelling is a critical process in software engineering and various other fields that involves creating simplified, abstract representations of complex systems. The primary purpose of system modelling is to understand, analyze, and communicate the structure, behaviour, and interactions of a system.

- *System modelling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system*
- *System modelling may represent a system using graphical notation, e.g. the Unified Modeling Language (UML).*

Graphical Modeling

Graphical modelling is a method of representing information, systems, or processes using visual elements such as diagrams, charts, graphs, and symbols. It is a way to convey complex concepts, relationships, and structures in a more intuitive and visual format.

The following graphical representations (SE Diagrams) are used for system modelling in software engineering.

1. **Activity Diagrams:** These diagrams depict the activities and processes involved in a system or data processing, showing the flow of tasks and their relationships.
2. **Use Case Diagrams:** Use case diagrams illustrate the interactions and relationships between a system and its external environment, typically representing actors and the actions they can perform.
3. **Sequence Diagrams:** Sequence diagrams are used to display interactions and sequences of messages between actors, objects, or system components over time, illustrating how various parts of a system work together.
4. **Class Diagrams:** Class diagrams provide a visual representation of object classes within a system and the relationships (associations) between these classes, helping to model the structure of a system.
5. **State Diagrams:** State diagrams depict how a system responds to internal and external events, displaying the different states a system or object can be in and the transitions between these states.

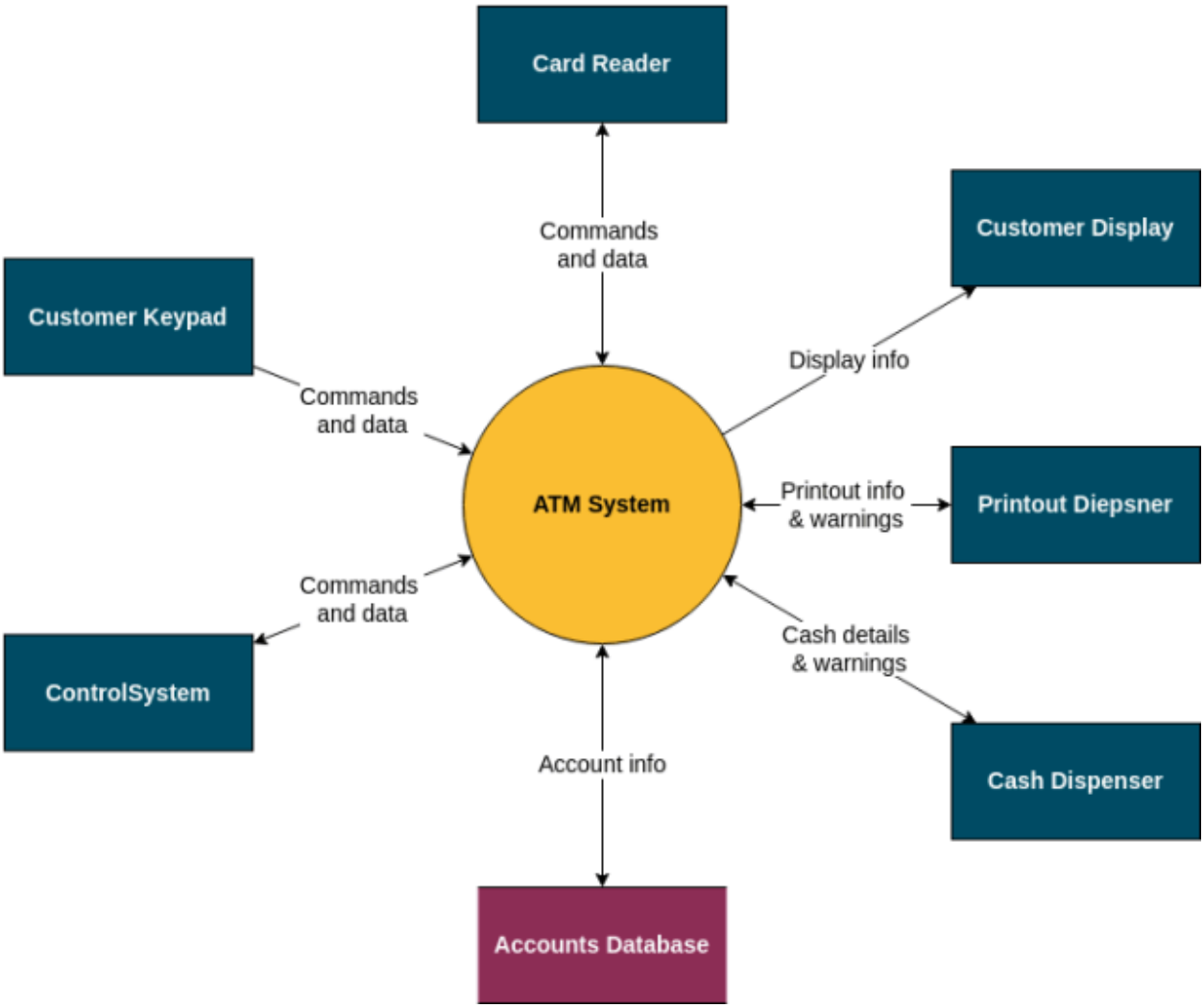
System Models in Software Engineering

From the system perspective, there are four types of models

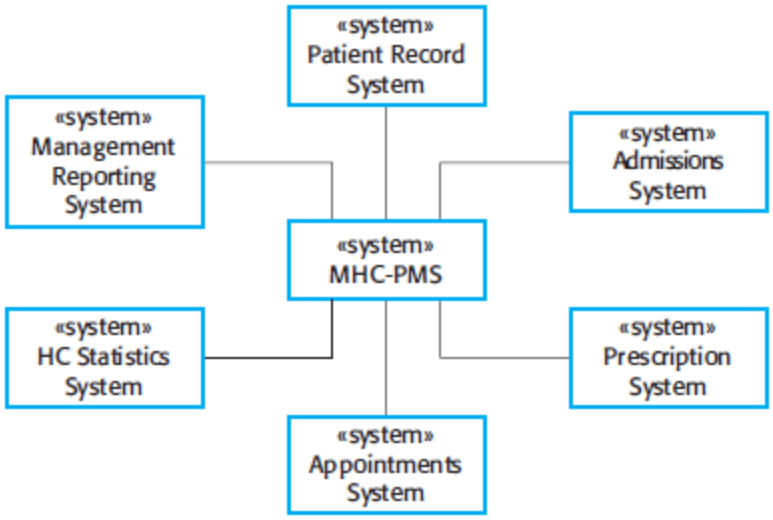
- Context models
- Interaction models
- Structural models
- Behavioural models
- Model-driven engineering

Context Models

- **Definition:** Context models provide a high-level representation of a system's environment and its interactions with external entities. They establish the system's boundaries and illustrate the relationships with outside entities.
- **Example:** Consider an automated teller machine (ATM) system. In a context model, you would depict the ATM as the system, and external entities could include the bank's database, bank customers, and the network. Arrows connecting these entities to the ATM would represent actions such as "Withdraw Cash" and "Check Balance," showing the system's interactions with external elements.



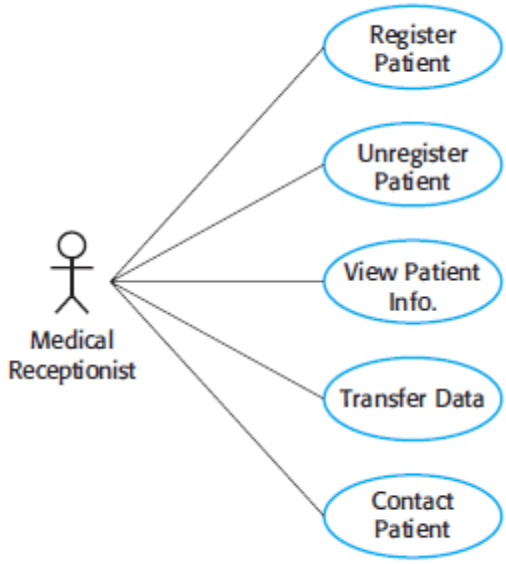
Context Model for ATM System



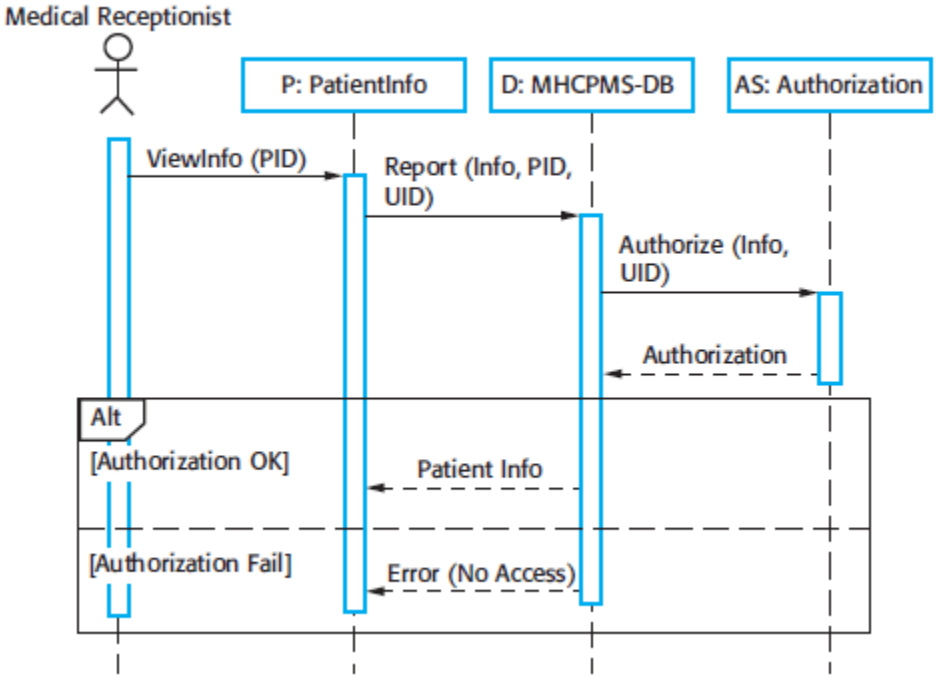
Context Model for Hospital

Interaction Models

- **Definition:** Interaction models focus on the dynamic aspects of a system by illustrating how its components or entities interact and exchange data. They often involve diagrams or sequence charts to represent the flow of information.
- **Example:** In an ATM example, how a user communicates (interacts) with the ATM (system) to withdraw money. A typical ATM interaction begins with the user inserting their card, entering their PIN, and selecting a transaction like a cash withdrawal. The ATM verifies the PIN, checks the account balance, dispenses cash, and offers the option to print a receipt. Once the transaction is complete, the user retrieves their cash and card, ending the interaction.



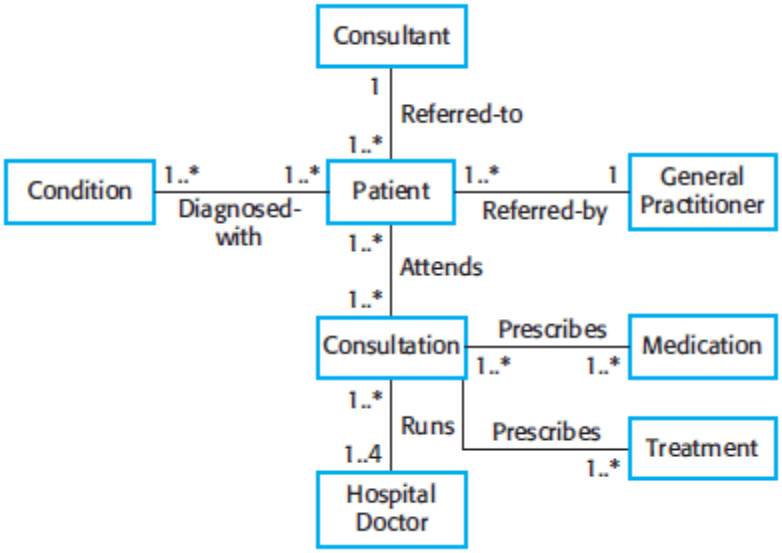
Receptionist interaction with the system (use case)



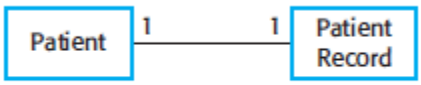
Sequence Diagram for Interaction

Structural Models

- **Definition:** Structural models focus on the static aspects of a system, emphasizing its components, their organization, and their relationships. They help to define the system's architecture.
- **Example:** This class diagram describes the structure of an ATM system. Each class has characteristics and functions. The bank class manages the ATM and can have one or many customers. Each customer can own zero or many debit cards and one account. The ATM transactions include both withdrawals and transfers. Each ATM transaction is modified by the account and identified by the ATM.



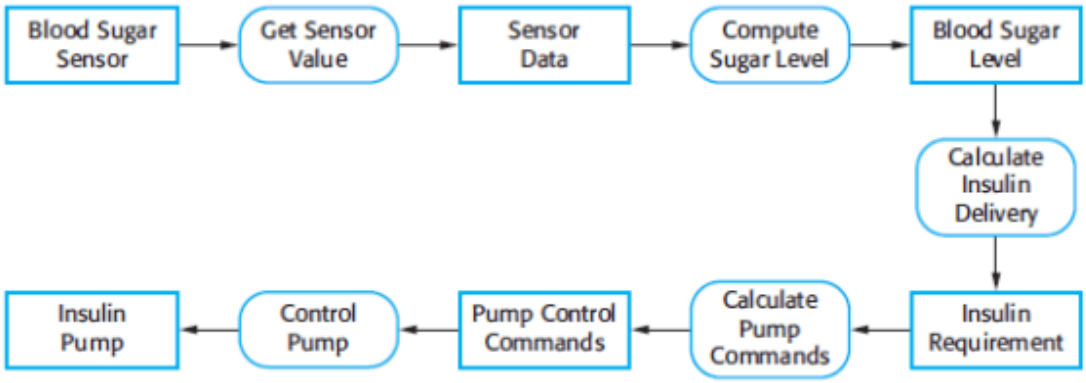
Structural model (class diagram) for patient management system



Class Diagram

Behavioural Models

- **Definition:** Behavioral models describe how a system functions over time, emphasizing the system's responses to various inputs or events. They help in understanding the system's dynamic behaviour.
- **Example:** Imagine an elevator system in a building. A behavioural model would include state diagrams showing how the elevator transitions between states (e.g., moving, stopping, opening doors, closing doors) based on user input and sensor data. This illustrates the dynamic behaviour of the elevator system.

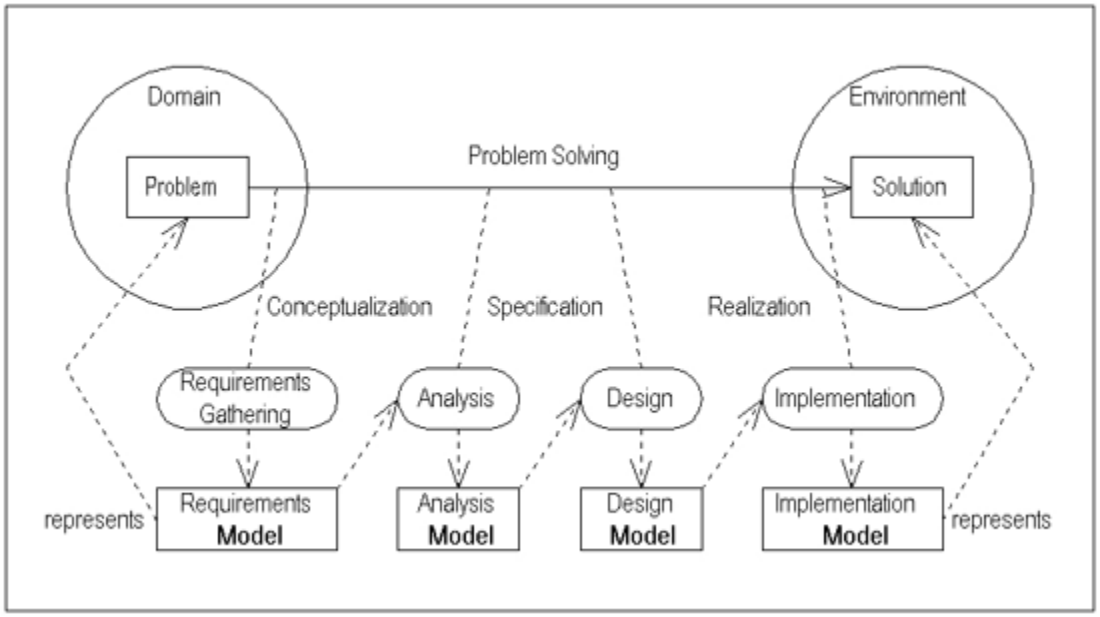


Activity Diagram for Behavioral Model

Model-driven engineering

Model-driven software engineering provides an approach such that one can design complex software or system in the form of a model and use that model to generate a code for the system automatically. As the model is a preferred output of model-driven software engineering (MDSE) there is no need for engineers to be concerned with the actual specifics of the execution platform or which programming language has to be used, there is no need for programming

language details. It allows engineers to think about software at a high level of abstraction, without being concerned about the implementation.



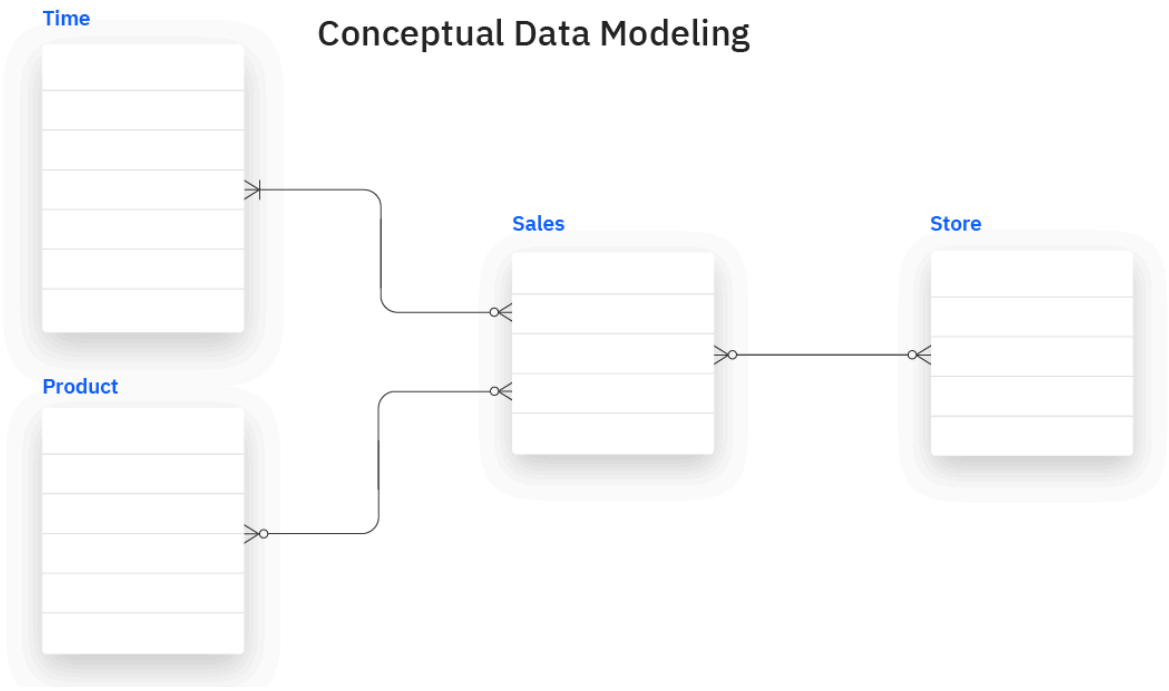
Data Model

Definition: Data modeling is the process of creating a visual representation of either a whole information system or parts of it to communicate connections between data points and structures.

Types of data models

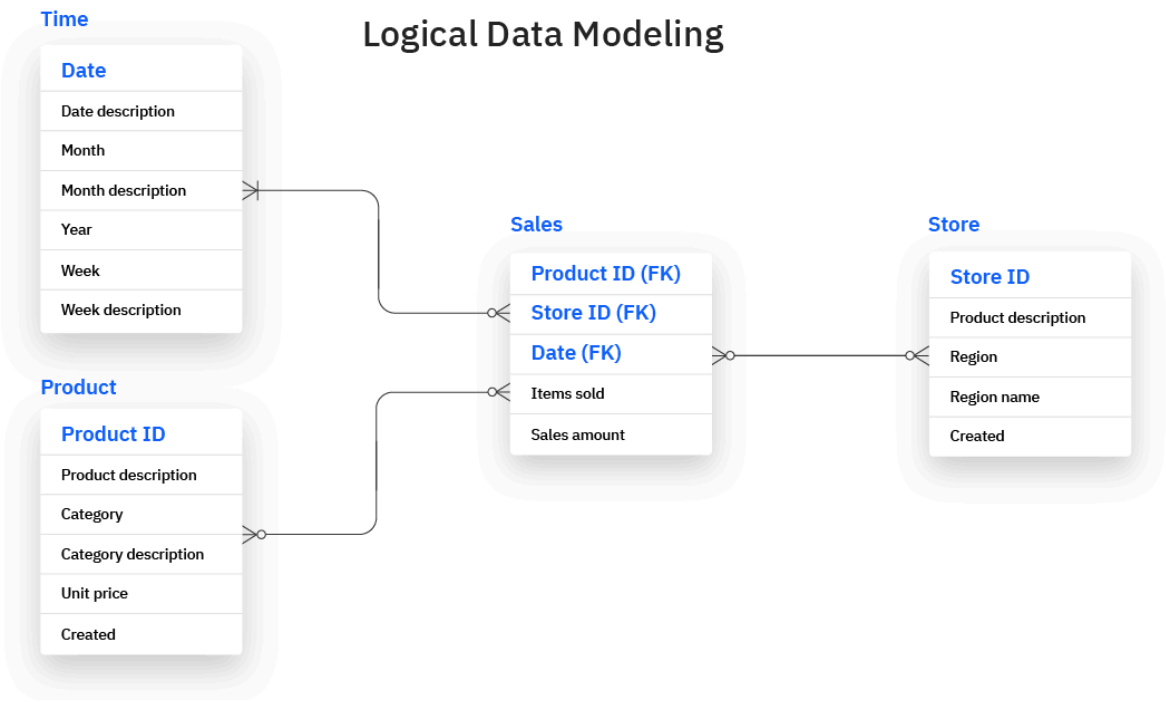
Conceptual data models

They are also referred to as domain models and offer a big-picture view of what the system will contain, how it will be organized, and which business rules are involved. Conceptual models are usually created as part of the process of gathering initial project requirements.



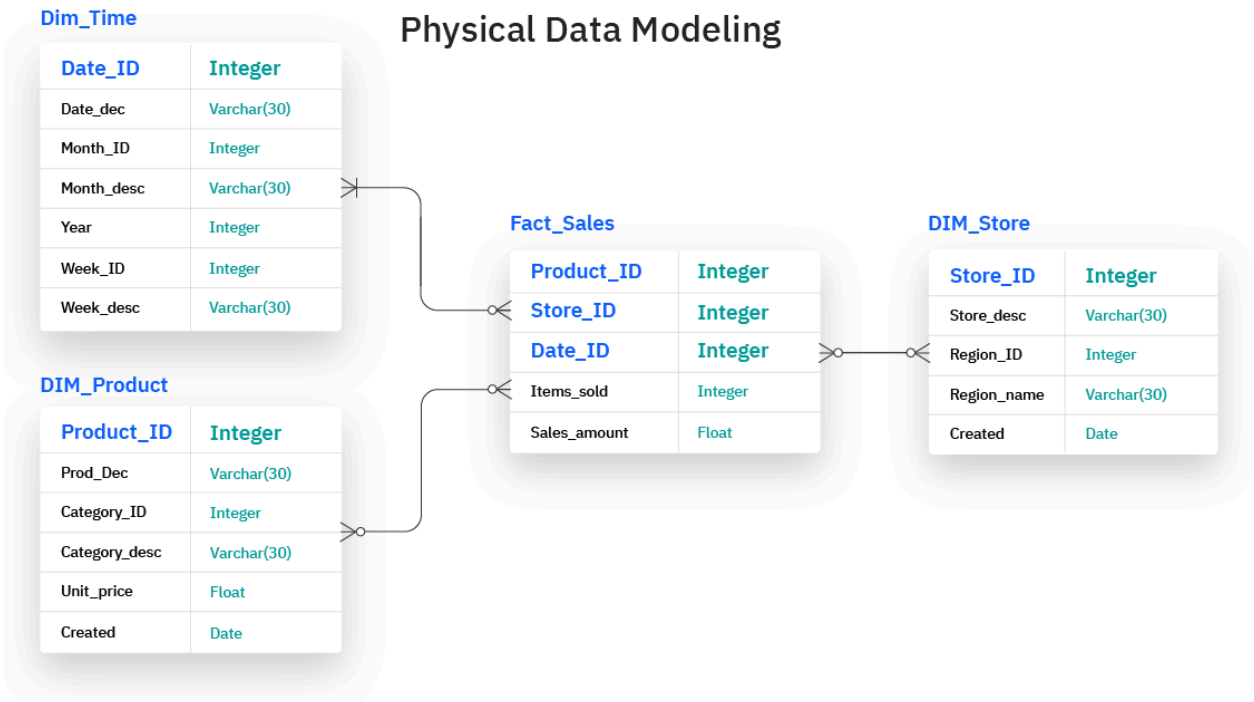
Logical data models

They are less abstract and provide greater detail about the concepts and relationships in the domain under consideration. One of several formal data modeling notation systems is followed. These indicate data attributes, such as data types and their corresponding lengths, and show the relationships among entities. Logical data models don’t specify any technical system requirements.



Physical data models

They provide a schema for how the data will be physically stored within a database. As such, they're the least abstract of all. They offer a finalized design that can be implemented as a relational database, including associative tables that illustrate the relationships among entities as well as the primary keys and foreign keys that will be used to maintain those relationships.



Object Model

In software engineering, an object model represents the structure of a system using objects, classes, and their relationships. It focuses on the static aspects of a system, defining the objects, their attributes, operations, and how they interact. Essentially, it's a blueprint that outlines the key components and their connections within a software system.

Objects:

These are instances of classes, representing real-world entities or concepts within the system. For example, in a banking system, objects could be "Account," "Customer," or "Transaction".

Classes:

These are blueprints for creating objects. They define the attributes (data) and operations (methods) that objects of that class will possess.

Attributes:

These are characteristics or properties of an object, like a customer's name, account number, or transaction amount.

Operations (Methods):

These are actions that an object can perform, such as depositing money into an account or withdrawing funds.

Relationships:

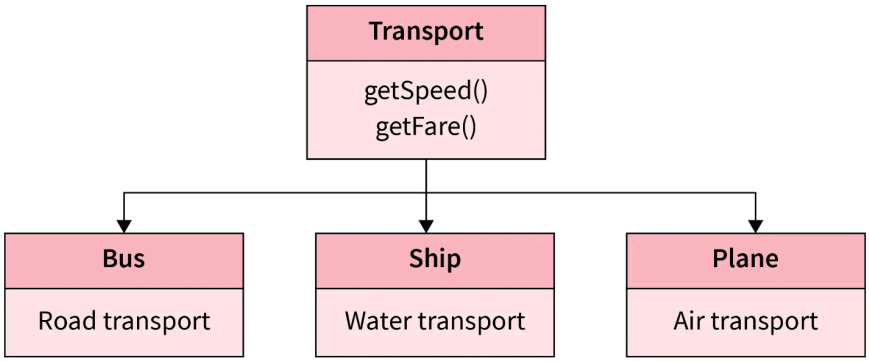
These define how objects interact with each other. Common relationships include:
Association: A general relationship between two classes, like a customer having one or more accounts.

Aggregation:

A part-of relationship, where one class is composed of other classes, such as a car being composed of an engine, wheels, etc.

Generalization (Inheritance):

A parent-child relationship where a subclass inherits attributes and operations from a superclass.



SCALER Topics