UNIT II

REQUIREMENTS AND SOFTWARE DESIGN

Introduction to Requirements engineering: Functional and Nonfunctional requirements – Requirement specification template – Eliciting requirements – Requirements analysis – Requirements modeling: Class-based modeling, Flow-oriented model, Behavioral model – Design process – Design concepts – Design model dimensions – Software architecture – Architectural styles – Architectural mapping using data flow – User interface analysis and design

2.1 INTRODUCTION TO REQUIREMENTS ENGINEERING: FUNCTIONAL AND NON FUNCTIONAL REQUIREMENTS

Requirements Engineering Key Concepts:

1. **Requirements Engineering**: This is a structured approach to understanding what the customer needs, analyzing those needs, negotiating solutions, and ensuring those solutions are clearly defined and managed throughout the development process. It bridges the gap between understanding the problem and actually designing a solution.
2. **Tasks in Requirements Engineering**: The passage describes **seven distinct tasks** involved in requirements engineering:
    - **Inception**: The early phase where the business need is identified, and a preliminary understanding of the project is formed.
    - **Elicitation**: Gathering information from stakeholders about what they need from the software. This can be challenging due to scope, understanding, and volatility issues.
    - **Elaboration**: Refining the initial requirements and creating detailed models based on user scenarios and analysis classes.
    - **Negotiation**: Resolving conflicts and prioritizing requirements among different stakeholders.
    - **Specification**: Documenting the requirements clearly, often using a combination of written text, diagrams, and models.
    - **Validation**: Ensuring the requirements are correct, consistent, and complete through reviews and feedback.
    - **Requirements Management**: Handling changes to requirements throughout the project lifecycle and ensuring they are properly tracked.
3. **Challenges in Requirements Engineering**: Several issues complicate this process:

- **Scope problems**: Ill-defined boundaries or unnecessary details that confuse the overall goals.
- **Understanding problems**: Stakeholders may not fully know what they need or might omit essential information.
- **Volatility**: Requirements may change as the project progresses, leading to the need for ongoing management and flexibility.

4. **Communication and Stakeholder Involvement**: Success in requirements engineering depends heavily on communication between the software team and stakeholders. The process starts with a rough understanding of the business case and evolves as more details are gathered, refined, and negotiated.

Why It's Important:

Requirements engineering serves as the foundation for successful software development. If done correctly, it ensures the software meets the needs of the customer, is feasible to develop, and provides a clear direction for the design and construction phases. Misunderstanding or ignoring requirements can lead to costly mistakes, scope creep, or an end product that doesn't meet user needs.

Although the process can seem tedious, especially when working with changing or unclear requirements, it ultimately ensures that the software being developed will be useful, usable, and deliver value to stakeholders. It requires careful attention, collaboration, and adaptability from everyone involved.

1. Functional Requirements:

These define **what** the system should do, focusing on the specific behaviors, functions, or services the system must provide. They describe the interactions between the system and its environment or users.

Characteristics of Functional Requirements:

- **Specific**: They define specific actions or tasks the system must be able to perform.
- **Clear and precise**: They should leave no room for ambiguity.
- **Verifiable**: They can be tested and confirmed by reviewing the system behavior.

Examples of Functional Requirements:

- The system must allow users to log in using their email and password.
- The system should calculate the total price of an order after the customer selects items.
- The application must allow users to search for products by category, price, and rating.

2. Nonfunctional Requirements:

These define **how** the system performs its functions, focusing on attributes like performance, reliability, security, usability, etc. They specify the constraints or conditions under which the system must operate, ensuring that the system functions correctly under various situations.

Characteristics of Nonfunctional Requirements:

- **Quality-focused**: These requirements relate to the quality or performance of the system.
- **Broad in scope**: They often apply to the overall system rather than individual features.
- **Measurable**: They can be evaluated and tested through specific metrics or standards.

Examples of Nonfunctional Requirements:

- The system must handle up to 1,000 concurrent users without performance degradation.
- The system should have an uptime of 99.9%.
- The application must be usable by people with disabilities, adhering to WCAG 2.1 standards.
- The system should encrypt all sensitive user data during transmission.

Key Differences Between Functional and Nonfunctional Requirements:

Aspect	Functional Requirements	Nonfunctional Requirements
Focus	Describes the system's functionality	Describes system qualities or constraints
Nature	Defines specific actions and tasks	Defines how well or under what conditions the system should perform
Measurement	Can be directly tested by system behaviour	Measured through system performance, security, etc.
Examples	Login, search, payment processing	Performance, security, scalability

Why Are Both Important?

- **Functional Requirements** ensure the system does what it's supposed to do, addressing the core business needs.
- **Nonfunctional Requirements** ensure the system works well under real-world conditions and satisfies user expectations regarding reliability, performance, and security.

Together, these requirements guide the development and testing processes, helping to ensure that the final product is functional, efficient, secure, and user-friendly.

2.2 REQUIREMENT SPECIFICATION TEMPLATE

A **Software Requirements Specification (SRS)** is a comprehensive document that describes the functional and nonfunctional requirements of a software system. It acts as a bridge between the stakeholders (like customers, users, and project managers) and the development team. The SRS outlines what the system is supposed to do, how it should perform, and any constraints or conditions it must operate under.

A *software requirements specification* (SRS) common template

Table of Contents

6. Other Requirements
Appendix A: Glossary
Appendix B: Analysis Models
Appendix C: Issues List

2.3 ELICITING REQUIREMENTS

Key Concepts:

1. **Collaborative Requirements Gathering**:
   This approach involves collaboration among software engineers and other stakeholders to identify, propose, negotiate, and specify the requirements of the software. It encourages active participation and teamwork in defining the system's objectives.
   - **Guidelines for Collaborative Gathering**:
     - **Meetings**: Stakeholders from both the software team and other relevant areas (e.g., business managers, users) are involved.
     - **Rules and Preparation**: Clear guidelines and agendas are set to ensure effective communication.
     - **Facilitator Role**: A neutral facilitator guides the meeting to ensure discussions stay on track.
     - **Definition Mechanisms**: Tools like worksheets, charts, or digital forums are used to document and refine the requirements.
2. **Product Request and Initial Lists**:
   Before the meeting, stakeholders review a **product request** document that outlines high-level goals (e.g., the home security function in the SafeHome example). Each participant is asked to prepare:
   - **Lists of objects**: Items in the system's environment (e.g., sensors, control panels).
   - **Services**: Functions or processes interacting with the objects (e.g., monitoring sensors, setting the alarm).
   - **Constraints**: Business, technical, or operational limitations (e.g., cost, compatibility).
   - **Performance Criteria**: Required system behaviors (e.g., speed, accuracy).

   These lists are then combined and refined during the meeting, ensuring all viewpoints are captured while avoiding critique during initial presentations.

3. **Mini-Specifications**:
   For clarity, some objects or services are elaborated further into **mini-specifications**. For example, a control panel mini-spec might describe its size, functionality, and user interface details. These mini-specs are discussed and refined with stakeholder input.

4. **Quality Function Deployment (QFD)**:
   QFD is a method used to ensure customer satisfaction by aligning the software requirements with customer needs. It categorizes requirements into three types:
   - **Normal Requirements**: Explicitly stated goals that satisfy customer needs (e.g., specific features).
   - **Expected Requirements**: Implicit but essential needs that customers take for granted (e.g., usability, reliability).
   - **Exciting Requirements**: Unexpected features that delight customers (e.g., novel functionalities).

   QFD tools, like customer interviews and surveys, are used to extract these requirements and prioritize them.

5. **Usage Scenarios**:
   Use cases or **usage scenarios** provide real-world examples of how users will interact with the system. These scenarios help bridge the gap between business requirements and technical design by clarifying the system's functionality in action. They provide a detailed understanding of how end-users will engage with the system.
6. **Elicitation Work Products**:
   The outcomes of requirements elicitation include:
   - A **statement of need and feasibility**, which outlines the core purpose and feasibility of the project.
   - A **bounded statement of scope**, detailing what the system will and will not include.
   - A list of **stakeholders** involved in the process.
   - A description of the **technical environment** the system will operate in.
   - A **list of functional requirements** and the **domain constraints**.
   - **Usage scenarios** that illustrate how the system will be used in various situations.
   - **Prototypes** (if applicable) to visually define or clarify requirements.

Flow of the Process:

The **requirements gathering meeting** follows a structured flow:

1. The **product request** is shared in advance to provide context.
2. Stakeholders create and review their individual lists of objects, services, constraints, and performance criteria.
3. These lists are then merged and discussed in a collaborative session, where ambiguities and conflicts are addressed.
4. The meeting may result in **mini-specifications** for further clarification.
5. An **issues list** is created for unresolved matters to be addressed later.

Importance of Collaboration:

The passage emphasizes that successful **requirements elicitation** relies on active collaboration and clear communication between all involved parties. This ensures that the final system will accurately reflect the needs and constraints of the users and stakeholders.

SUMMARY

Requirements elicitation is a critical phase in software development that sets the foundation for the entire project. By combining collaborative techniques, clear documentation, and structured approaches like **Quality Function Deployment** and **use cases**, teams can ensure they gather comprehensive and accurate requirements. This phase helps prevent miscommunication, scope creep, and costly errors in the later stages of development.

Requirements Elicitation Overview:

- **Definition**:
  Requirements elicitation (or gathering) is a process that combines problem-solving, elaboration, negotiation, and specification to collaboratively define the needs and expectations for a system.

Collaborative Requirements Gathering:

1. **Purpose**:
   To create a shared understanding of the system's requirements by involving both software engineers and stakeholders in the process.
2. **Basic Guidelines**:
   - **Participants**: Software engineers and other stakeholders.
   - **Preparation**: Clear rules for preparation and participation.
   - **Agenda**: Balanced between formal and informal to encourage idea flow.
   - **Facilitator**: A neutral party (e.g., customer, developer, outsider) to manage the meeting.
   - **Definition Mechanism**: Tools like worksheets, flip charts, or digital boards to record and organize ideas.
3. **Process Flow**:
   - **Pre-meeting**: A product request (e.g., SafeHome project) is written, shared, and reviewed.
     - Example: SafeHome product request describes home security features.
   - **List Creation**: Attendees create lists of:
     - **Objects**: Items in the system's environment (e.g., sensors, alarms).
     - **Services**: Functions that interact with the objects (e.g., monitoring, dialing the phone).
     - **Constraints**: Technical, business, and environmental limits (e.g., cost, compatibility).
     - **Performance Criteria**: System behaviors (e.g., speed, accuracy).
   - **Review & Discussion**:
     - **Combine lists**: Merge similar ideas, add new concepts.
     - **Mini-Specifications**: Elaborate on complex objects or services.
     - **Issues List**: Track unresolved items for later action.

Quality Function Deployment (QFD):

1. **Purpose**:
   A technique to convert customer needs into technical requirements that drive software development. QFD aims to maximize customer satisfaction.
2. **Types of Requirements**:
   - **Normal Requirements**: Explicit goals that ensure customer satisfaction (e.g., requested features or performance).
   - **Expected Requirements**: Implicit expectations that, if missing, would cause dissatisfaction (e.g., reliability, usability).
   - **Exciting Requirements**: Unexpected features that delight customers (e.g., innovative functionalities).
3. **QFD Process**:
   - Gather data from customer interviews, surveys, and historical data (e.g., past problem reports).
   - Create a **customer voice table** to capture customer needs.
   - Use diagrams, matrices, and evaluations to analyze and derive requirements.

Usage Scenarios (Use Cases):

1. **Purpose**:
   To understand how different users will interact with the system.
2. **Benefits**:
   - Helps clarify system functions and features.
   - Provides a context for design by identifying typical user interactions.
3. **Output**:
   A set of **use cases** or scenarios that describe how end-users will engage with the system.

Elicitation Work Products:

1. **Work Products (Outputs of Requirements Elicitation)**:
   - **Statement of Need & Feasibility**: Defines the problem and whether the project is feasible.
   - **Bounded Statement of Scope**: Outlines what is included and excluded from the system.
   - **Stakeholder List**: Names of all participants involved in the process.
   - **System's Technical Environment**: Describes the environment in which the system will operate.
   - **List of Requirements**: Organized by function, with domain constraints.
   - **Usage Scenarios**: Describes system use under different conditions.
   - **Prototypes**: Visual or functional mock-ups that help define requirements more clearly.
2. **Review**:
   All work products should be reviewed by all stakeholders involved in the elicitation process.

Summary of Collaborative Requirements Gathering Process:

- Gather initial information through product requests.
- Stakeholders create and discuss lists of objects, services, constraints, and performance criteria.
- These lists are combined, clarified, and refined through discussions and mini-specifications.
- QFD and usage scenarios are used to ensure customer needs are captured accurately and understood by all participants.

2.4 REQUIREMENTS ANALYSIS

This excerpt provides an overview of the requirements analysis and modeling process, offering insights into the objectives, principles, and approaches to creating a software requirements model. Below is a summarized breakdown of the key points discussed:

Key Concepts in Requirements Analysis and Modeling

1. **Objectives of Requirements Analysis:**
    - **What needs to be done:** Focus on understanding *what* the system must do rather than *how* it will do it.
    - **Iterative process:** Since complete requirements may not be fully known, the analysis evolves iteratively. Models are created based on what is known at the time, allowing for feedback and adjustments.
    - **Three primary goals:**
        1. **Describe customer requirements:** Accurately capturing what the customer needs.
        2. **Create the foundation for design:** Provide enough detail to guide software design.
        3. **Validation:** Ensure that the model can be validated once the system is built.
2. **Requirements Modeling:**
    - **Focus areas:** Different types of models are created to represent the system's functional, informational, and behavioral aspects:
        - **Scenario-based models**: Show interactions from the user's perspective.
        - **Data models**: Represent the information domain of the system.
        - **Class-oriented models**: Define object-oriented classes and their collaborations.
        - **Flow-oriented models**: Focus on how data flows through the system.
        - **Behavioral models**: Depict system behavior in response to external events.
    - **Objective:** To provide sufficient detail for system design while ensuring that all components are traceable to the design.
3. **Rules of Thumb for Effective Analysis Modeling:**

- Focus on *high-level abstraction* and avoid getting bogged down in implementation details.
- Ensure that each element in the model contributes to a better understanding of the system's information, function, and behavior.
- Delay considering technical infrastructure (e.g., databases) until later in the design phase.
- Minimize unnecessary dependencies between system components (coupling).
- Ensure that all stakeholders (business, design, QA) benefit from the model.
- Keep the model as simple as possible, using the least amount of information necessary to communicate the requirements.

4. **Domain Analysis:**
   - The process of identifying and specifying reusable patterns, objects, and classes that apply across multiple projects within a particular domain (e.g., banking, medical devices).
   - It focuses on discovering common requirements and capabilities that can be reused in future projects, improving development efficiency and reducing costs.
   - Domain analysis is an ongoing activity that ensures the creation of reusable analysis classes and patterns, contributing to faster development cycles.
5. **Modeling Approaches:**
   - **Structured analysis**: Emphasizes the data and processes that transform data in the system.
   - **Object-oriented analysis (OOA)**: Focuses on defining classes and their relationships and interactions to meet requirements.
   - A hybrid approach combining both structured and object-oriented methods can provide a more comprehensive understanding of requirements.
6. **Analysis Models:**
   - **Scenario-based elements**: Illustrate user-system interactions.
   - **Class-based elements**: Describe the system's objects, their attributes, operations, and relationships.
   - **Behavioral elements**: Show how the system's state changes in response to events.
   - **Flow-oriented elements**: Represent the transformation of data as it flows through the system.

Domain Analysis and Reuse:

- Domain analysis can be compared to creating reusable tools for software projects. The domain analyst's role is to identify reusable classes, patterns, and objects that can be shared across multiple projects in a particular domain.
- The **goal** of domain analysis is to streamline development processes by providing reusable components that save time and reduce costs.

Best Practices:

- Avoid creating unnecessary or overly complex diagrams that do not provide additional value.
- Choose the right combination of modeling techniques based on the project's needs and stakeholder requirements.

- Ensure that the requirements model provides clarity for all stakeholders, supporting design, validation, and testing processes.

Effective requirements analysis and modeling provide the foundation for building quality software by clearly defining what the software should do, how it should behave, and what constraints it must operate within. A clear, simple, and iterative approach to analysis allows for better understanding, easier communication among stakeholders, and a more efficient transition to software design and development.

2.5 REQUIREMENTS MODELING: CLASS-BASED MODELING, FLOW-ORIENTED MODEL, BEHAVIORAL MODEL

2.5.1 Class-Based Modeling in Software Engineering:

Key Concepts in Class-Based Modeling

Class-based modeling is a core part of object-oriented software design. It involves mapping real-world entities and their interactions into software system components. The main elements include **objects**, **attributes**, **operations**, **relationships**, and **collaborations** between classes. These models provide a structure to translate requirements into a functional design.

1. Identifying Analysis Classes

The process begins with identifying potential classes that represent real-world entities or concepts relevant to the system. These are often discovered through **use cases**, focusing on **nouns** or **noun phrases** that describe entities, roles, events, and organizational units.

Examples of potential classes in a SafeHome system:

- **External entities**: Homeowner, Monitoring Service.
- **Things**: Sensors, Alarm.
- **Events**: Sensor Event, System Activation.
- **Roles**: Homeowner, Technician.
- **Places**: Control Panel, Installation Site.
- **Structures**: System, Sensor.

To evaluate whether a noun should be a class, Coad and Yourdon's six **selection characteristics** are used:

1. **Retained Information**: Does the class hold data that needs to be remembered?
2. **Needed Services**: Does it have identifiable operations to manipulate its data?
3. **Multiple Attributes**: Does the class have several defining attributes?

4. **Common Attributes**: Can common attributes apply to all instances of the class?
5. **Common Operations**: Are there shared operations for all instances?
6. **Essential Requirements**: Is the class essential in fulfilling system requirements?

2. Specifying Attributes

Attributes define the **properties** of a class and describe the data each instance will hold. Attributes should logically relate to the class's purpose.

Examples for the System class in SafeHome:

- Identification: `systemID`, `verificationPhoneNumber`, `systemStatus`.
- Alarm response: `delayTime`, `telephoneNumber`.
- Activation/deactivation: `masterPassword`, `numOfAllowableTries`, `temporaryPassword`.

Attributes should be meaningful, ensuring that only relevant information is encapsulated within the class.

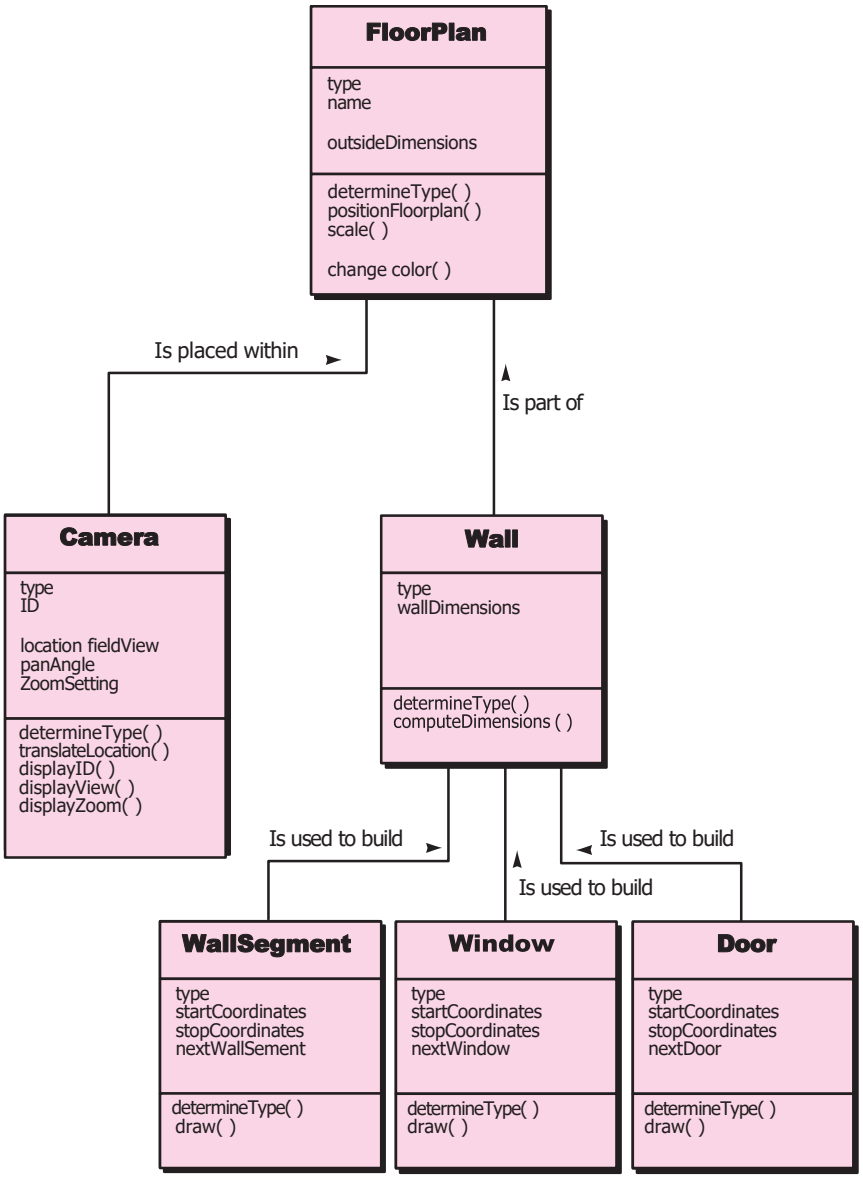

FIGURE 2.1:CLASS DIAGRAM

3. Defining Operations

Operations (or **methods**) define the **behavior** of the class. These are the actions or services that a class can perform, typically derived from verbs in the use cases.

Examples of operations:

1. **Data manipulation**: `assign()`, `program()`.
2. **Computation**: `calculate()`, `determineValue()`.
3. **State inquiries**: `isArmed()`, `isActivated()`.
4. **Event monitoring**: `monitorSensors()`, `triggerAlarm()`.

For instance:

- **Sensor class** may have `assign()` to assign a sensor a type or number.
- **System class** may have `arm()` and `disarm()` to control system activation.

4. Collaboration Between Objects

Once classes, attributes, and operations are defined, the next step is to explore how objects interact. Objects communicate through **messages**, triggering operations in other objects. **Collaboration diagrams** or **CRC (Class-Responsibility-Collaborator) cards** help visualize these interactions.

Class-Responsibility-Collaborator (CRC) Modeling

CRC modeling is a simple yet effective technique for organizing and structuring class relationships.

CRC Card Structure

1. **Class Name**: The name of the class (e.g., `Sensor`, `System`).
2. **Responsibilities**: Attributes and operations the class is responsible for.
3. **Collaborators**: Other classes that work together with this class to fulfill its responsibilities.

Example CRC for a `FloorPlan` class:

- **Responsibilities**: Represent building layout, interact with `Wall` and `Camera` classes for surveillance.
- **Collaborators**: `Wall`, `Camera`.

Types of Classes

1. **Entity Classes**: Directly extracted from the problem domain. They are persistent throughout the application's lifecycle (e.g., `Sensor`, `FloorPlan`).
2. **Boundary Classes**: Represent system interfaces for user interaction (e.g., `CameraWindow`).

3. **Controller Classes**: Handle processes involving multiple objects, such as creating entities or validating data (e.g., controllers defined in the design phase).

Guidelines for Allocating Responsibilities

- **Distribute System Intelligence**: Responsibilities should be spread out across multiple classes. Avoid overloading a single class.
- **Generalization**: Keep responsibilities at a higher level of abstraction for flexibility.
- **Encapsulation**: Data and behavior related to that data should reside in the same class.
- **Localization**: Group related data in one class for easier maintenance.
- **Shared Responsibilities**: Related classes can share responsibilities if they exhibit similar behavior.

Collaboration and Dependencies Between Classes

Classes interact through **collaborations** and **dependencies**. These are represented by various relationships:

- **Is-part-of**: One class is a part of another (e.g., `Wall` is part of `FloorPlan`).
- **Has-knowledge-of**: One class needs information from another to fulfill responsibilities (e.g., `ControlPanel` needs data from `Sensor`).
- **Depends-upon**: Classes are dependent on each other for functionality (e.g., `DisplayWindow` depends on `Camera` for video feed).

Associations and Dependencies

- **Associations**: Describe how two classes are related (e.g., a `Wall` object may have associations with `Window`, `Door`).
- **Dependency Relationships**: One class may rely on another for information or actions (e.g., `Camera` provides video data to `DisplayWindow`).

Analysis Packages

In large systems, grouping related classes into **analysis packages** helps maintain organization and clarity. An **analysis package** is a collection of related classes focusing on specific aspects of the system.

Example packages in a video game system:

- **Visual Elements**: `Tree`, `Road`, `Building`.
- **Characters**: `Player`, `Protagonist`, `Antagonist`.
- **Game Rules**: `RulesOfMovement`, `ConstraintsOnAction`.

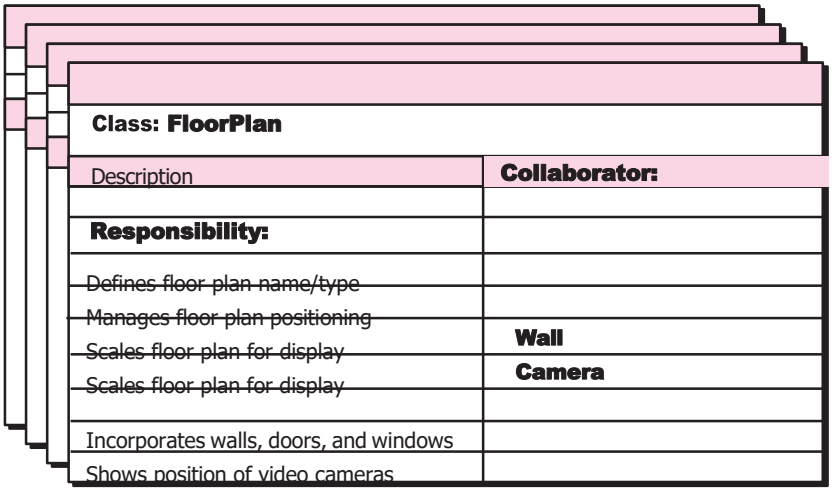

FIGURE 2.2 CRC MODEL INDEX CARD

Reviewing CRC Models

1. **Divide Cards**: Each participant in the review holds a subset of CRC cards.
2. **Use-Case Scenarios**: The review leader presents use cases, passing tokens to those with relevant CRC cards.
3. **Responsibilities Fulfillment**: Once a token reaches a collaborator class, the holder describes its role and interactions.
4. **Review and Adjust**: If any responsibilities or collaborations don't meet the use case, modifications are made.

Class-based modeling is a vital part of object-oriented design, helping to translate real-world requirements into functional system components. The process of defining classes, responsibilities, operations, and collaborations ensures that the software system reflects the problem domain accurately, is structured for maintainability, and supports efficient interaction between its components. CRC cards and the classification of entities, boundaries, and controllers streamline this process, allowing for better organization and clearer system design.

2.5.2 Flow-Oriented Modeling in Software Engineering:

Flow-oriented modeling is a key technique in software engineering that focuses on representing how data and control flow within a system. It uses **Data Flow Diagrams (DFDs)** and **Control Flow Models** to understand system requirements, especially for complex systems like SafeHome (a security system). These techniques help visualize both data processing and the system's behavior during state transitions, ensuring that the system's functionality is well understood before implementation.

---

1. Data Flow Diagrams (DFD)

Data Flow Diagrams (DFDs) are used to model the flow of data through a system, capturing the inputs, outputs, processes, and data stores involved in the system's operations.

Levels of DFD:

- **Level 0 (Context Diagram)**:
  - The system is shown as a single process (a "bubble") interacting with external entities (represented as "boxes").
  - The diagram depicts high-level interactions, such as data flowing into and out of the system.
  - This level provides an overview of the system.
- **Level 1**:
  - Breaks down the context diagram into more detailed processes (sub-bubbles).
  - This level specifies how data flows between processes, and the role of data stores in processing information.
- **Subsequent Levels**:
  - Further refinement of processes takes place.
  - As processes are decomposed, each level gets more specific, ensuring simplicity in implementation.

Guidelines for Creating a DFD:

1. **Start Simple**: Begin with a Level 0 DFD that represents the system as a whole.
2. **Identify Inputs and Outputs**: Focus on the key inputs and outputs.
3. **Refine in Steps**: Gradually break down processes, data objects, and stores.
4. **Label Meaningfully**: Each process, data flow, and store should be clearly labeled.
5. **Maintain Data Flow Continuity**: Ensure that data flows logically between levels, with consistency in how it's represented.

Example from SafeHome:

- **Level 0**: The DFD would show interactions between SafeHome (system) and external entities such as users and sensors.
- **Level 1**: The system would be broken down into processes like configuring the system, monitoring sensors, and user interactions.

2. Control Flow Modeling

Control flow modeling is essential for systems where the behavior depends on events or transitions between different states. It's particularly useful in **real-time**, **event-driven**, or **safety-critical** systems like SafeHome.

Creating a Control Flow Model:

Control flow models capture the system's behavior, defining how it transitions from one state to another based on specific events. Each state represents a mode or condition the system can be in, and events trigger transitions between these states.

Key Components:

- **Events**: External triggers like a sensor being activated, a user pressing a button, or a timer reaching a certain value.
- **States**: The operational modes of the system, such as "Idle", "Monitoring", "Triggered".

State Diagrams:

- State diagrams represent the system's sequential behavior by showing the states and transitions.
- Transitions are triggered by specific events, and the system's response is part of the transition.

Example in SafeHome:

- Transition from "Idle" to "Monitoring" when the system is activated.
- Transition back to "Idle" when deactivated.

3. Control Specification (CSPEC)

Control Specifications (CSPEC) describe the behavior of the system in terms of events and states. CSPEC includes both state diagrams and a **Program Activation Table (PAT)**.

- **State Diagrams**: Sequential representations of system behavior, showing states and events.
- **Program Activation Table (PAT)**: A table defining which processes are activated when specific events occur.

Example for SafeHome: The CSPEC would define how SafeHome responds to different sensor events or system commands (e.g., alarm triggers, system reset).

4. Process Specification (PSPEC)

The **Process Specification (PSPEC)** provides a detailed description of the behaviors of processes defined at the lower levels of the DFD. It is essential for translating the high-level model into specific, implementable details.

- **Narrative Text**: Describes the process functionality in detail.
- **Program Design Language (PDL)**: Provides more technical, algorithmic details, guiding the implementation.

Example for SafeHome: The PSPEC for the process "Password Validation" might describe how the system checks a user's password, first against a master password and then against secondary passwords.

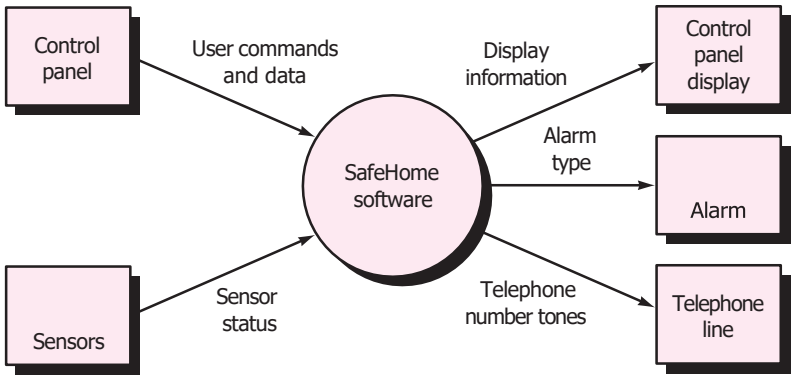

FIGURE 2.3 LEVEL 0 CONTEXT DIAGRAM FOR SAFE HOME

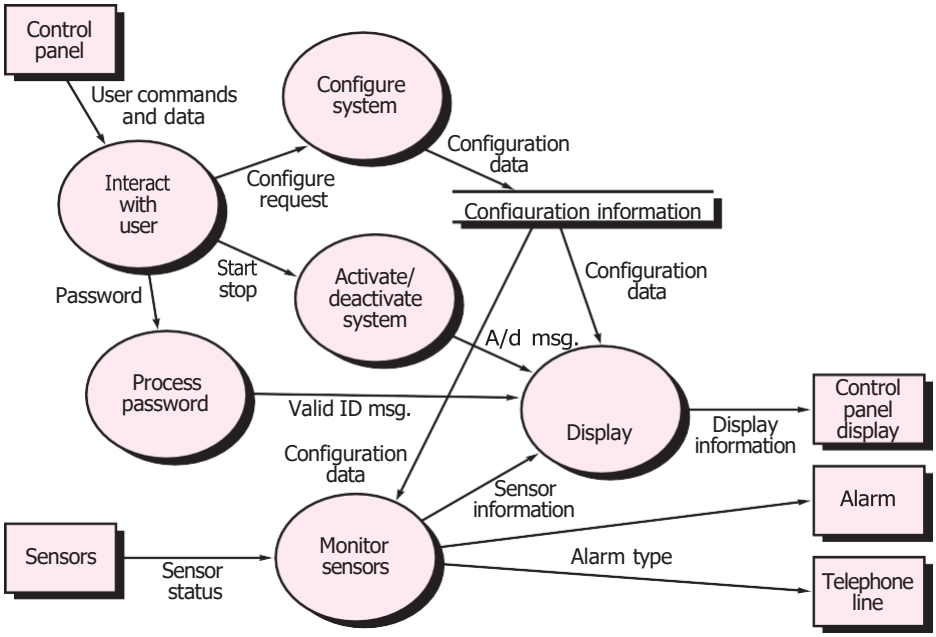

FIGURE 2.4 LEVEL 1 DFD DIAGRAM FOR SAFE HOME SECURITY FUNCTION

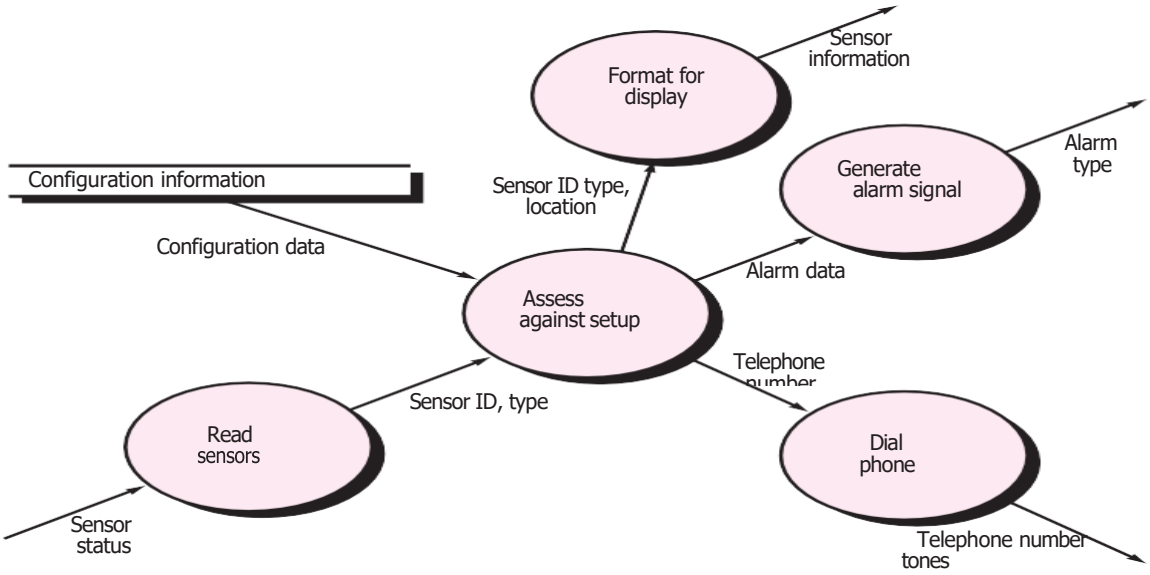

FIGURE 2.5 LEVEL 2 DFD FOR SAFE HOME SECURITY FUNCTION

5. Key Insights and Best Practices

To ensure that flow-oriented modeling is effective, here are some important insights and best practices:

1. **Data Flow Continuity**:
   - Maintain logical consistency as you refine your DFD. Each level of the DFD should build on the previous one with clear, consistent data relationships.
2. **Iterative Refinement**:
   - Both DFD and control flow models should be refined iteratively. Begin with high-level abstractions, then progressively break down into more detailed processes and states.
3. **System Behavior Focus**:
   - Control flow modeling is essential for capturing the dynamic behavior of event-driven systems. It helps model the transitions between states based on external events or internal conditions.
4. **State and Process Specifications**:
   - Detailed specifications at the lower levels of the DFD provide crucial guidance to the design phase, ensuring the system's operations can be effectively implemented.

Flow-oriented modeling, using **Data Flow Diagrams (DFDs)** and **Control Flow Modeling**, provides a comprehensive approach to understanding and designing systems. DFDs focus on data processing and interactions, while control flow diagrams emphasize event-driven behaviors and system state transitions. Together, they ensure that both data flow and system behavior are thoroughly understood, guiding software designers to develop a system that meets all the specified requirements.

By following best practices like maintaining data flow continuity, refining iteratively, and focusing on system behavior for event-driven systems, you can ensure the software design is robust, efficient, and aligned with user needs.

2.5.3 BEHAVIORAL MODEL

Creating Behavioral Modeling

Behavioral modeling is an essential part of software engineering that focuses on how the system reacts to events over time and how the system components interact. The goal is to represent the dynamic behavior of the system in response to different events, conditions, and user interactions. Below are the key steps involved in creating a behavioral model:

1. Identifying Events with the Use Case

Events drive the behavior of a system, and they are often triggered by external stimuli, such as user actions or system conditions. Identifying events within use cases is the first step in understanding how the system will behave. Here's how to approach identifying events:

Steps:

- **Analyze Use Cases**: A use case describes the interaction between a user (or other systems) and the system to achieve a particular goal. Start by reviewing each use case and understanding the sequence of actions involved.
- **Identify Key Actions**: Identify the actions or interactions that occur within the system. For example, if the use case is "Homeowner enters password into the keypad," the event is the "password entered."
- **Map Events to System Actions**: Each event should trigger a system action, such as validation or transitioning to a new state.

Example (Home Security System):

- **Use Case**: "Homeowner enters password to unlock the system."
  - **Event**: "Password entered."
  - **Actors**: Homeowner (initiates the event) → Control Panel (receives the event)
  - **Action**: The system compares the entered password with the stored password.

Additional Event Considerations:

- **Event Triggers**: Some events may be triggered by external systems (e.g., sensor activation).
- **Internal Events**: Events that happen within the system, such as timer expiration or a state change.
- **Output Events**: Responses that the system generates as a result of an event, such as a success message or an alert.

2. State Representations

State representations describe the different **states** in which an object or system can exist at any given time, as well as how the system transitions between states in response to events. Each state represents a particular condition or situation of the system.

Types of States:

- **Active State**: The system is engaged in processing or performing an action (e.g., "Validating password").
- **Passive State**: The system is idle or in a resting state, awaiting user input or external events (e.g., "Idle" state of the control panel).

Key Concepts for State Representation:

- **State Diagram**: A graphical representation that shows the states of a system or object, along with transitions between states based on events.
- **Transitions**: Arrows between states that show how the system moves from one state to another in response to events.
- **Triggers**: Events that cause the transition from one state to another (e.g., "Password entered").
- **Guard Conditions**: Conditions that must be met for a transition to occur (e.g., password length must be four digits).
- **Actions**: Operations performed as a result of a state transition (e.g., validating the password).

Example (Control Panel in Home Security System):

- **States**:
  - **Idle**: Waiting for user input.
  - **Reading**: User is entering the password.
  - **Validating**: System is comparing the entered password.
  - **Unlocked**: System grants access after correct password.
  - **Locked**: System denies access after incorrect password.
- **State Transitions**:
  - From **Idle** to **Reading** when a user enters the password.
  - From **Reading** to **Validating** when the system processes the entered password.
  - From **Validating** to either **Unlocked** or **Locked** depending on the result of the password comparison.

State Diagram Example:

- **State**: "Idle" → **Event**: "Password entered" → **Transition**: "Reading"
- **State**: "Reading" → **Event**: "Password validated" → **Transition**: "Validating"
- **State**: "Validating" → **Event**: "Valid password" → **Transition**: "Unlocked"
- **State**: "Validating" → **Event**: "Invalid password" → **Transition**: "Locked"

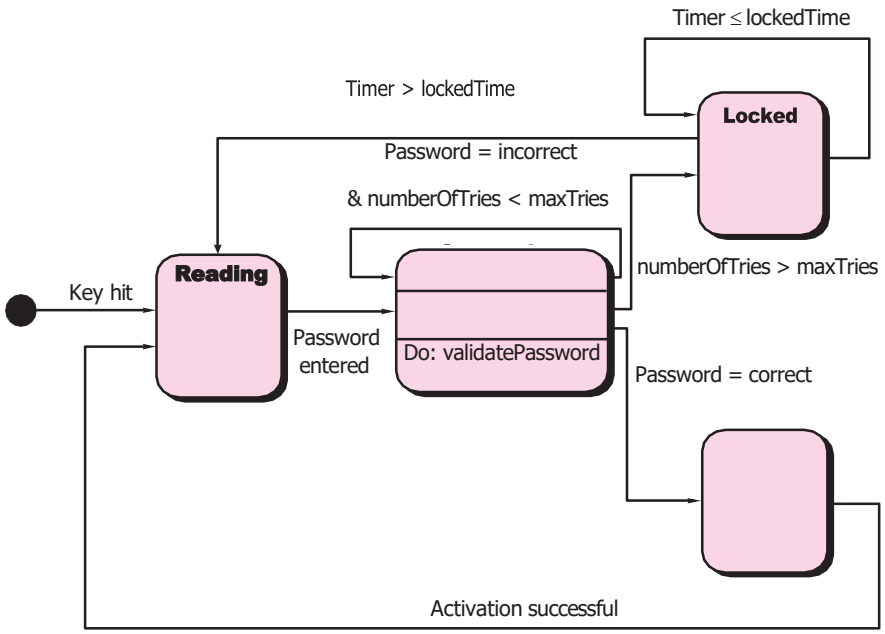

FIGURE 2.6 STATE DIAGRAM FOR CONTROL PANEL CLASS

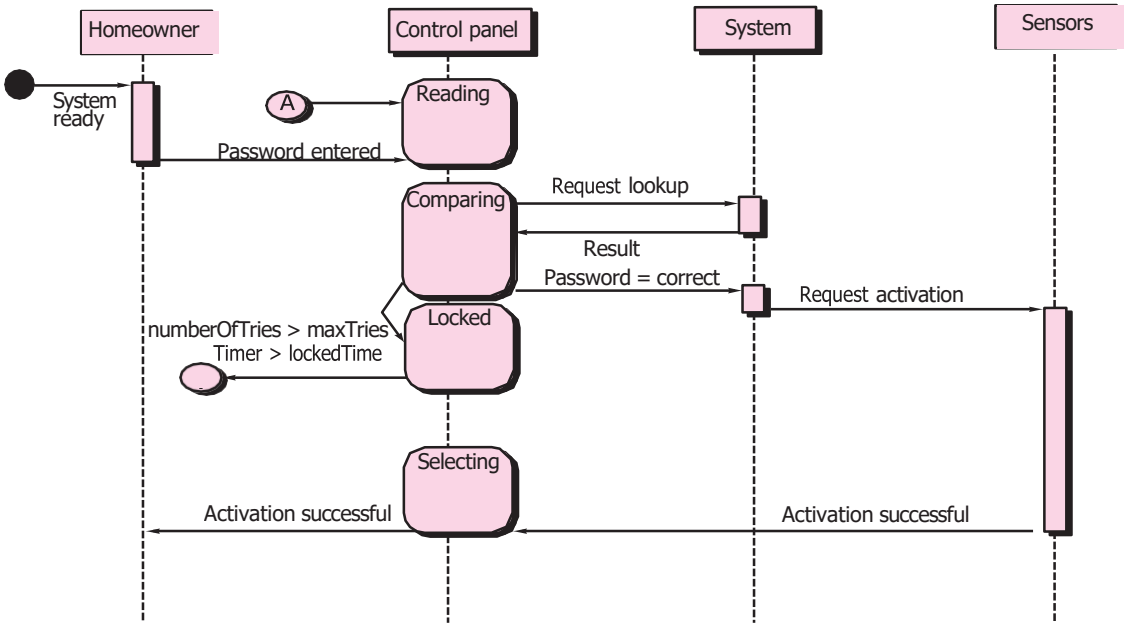

FIGURE 2.7 SEQUENCE DIAGRAM FOR SAFE HOME

3. Sequence Diagrams

Sequence diagrams are used to model how objects interact with each other over time in response to events. These diagrams help visualize the **flow of control** between objects, showing how information is exchanged step-by-step.

Key Components of Sequence Diagrams:

- **Objects**: The entities involved in the interaction, represented by vertical lifelines.
- **Lifelines**: Vertical dashed lines representing the objects' existence over time.
- **Messages**: Arrows between objects that represent communication or data exchange.
- **Activation Boxes**: Vertical rectangles representing the time an object spends performing an operation.
- **Events**: Actions or triggers that initiate a message or state change.

Steps to Create a Sequence Diagram:

1. **Identify the scenario**: Choose a specific use case or scenario you want to represent, such as "Password Validation."
2. **Identify involved objects**: Determine which objects or components are involved in the scenario (e.g., Homeowner, Control Panel, System).
3. **List the events**: Identify the events that trigger actions between objects (e.g., "password entered").
4. **Map interactions**: Draw arrows representing the sequence of interactions between objects and include the events that trigger each action.
5. **Add activation boxes**: Indicate the period during which each object is active (e.g., when the Control Panel is validating the password).

Example (Password Validation in Sequence Diagram):

1. **Event**: Homeowner enters password.
    - **Control Panel**: Receives the event "password entered" and triggers a validation.
2. **Event**: Control Panel sends "validate password" to System.
    - **System**: Compares the entered password with the stored password.
3. **Event**: System sends result (valid/invalid) to Control Panel.
4. **Action**: Based on the result, the Control Panel either grants access (unlocks) or denies access (locks).

Sequence Diagram Example:

- **Object 1**: Homeowner (initiates action)
- **Object 2**: Control Panel (receives input and sends validation request)
- **Object 3**: System (performs validation)

```
Homeowner --> Control Panel: Enter password
Control Panel --> System: Validate password
System --> Control Panel: Valid/Invalid result
Control Panel --> Homeowner: Access granted/denied
```

Creating behavioral models involves:

1. **Identifying events** from use cases and how they trigger actions within the system.
2. **Representing system states** and how the system transitions between states based on those events.
3. **Using sequence diagrams** to model interactions between system components over time, ensuring that events and their responses are accurately captured.

Behavioral modeling helps provide a comprehensive understanding of how the system behaves in various scenarios, ensuring that developers and stakeholders can anticipate how the system will perform and interact under different conditions.

2.6 The Design Process

The software design process is **iterative**, where initial high-level designs are progressively refined and made more detailed over time. Initially, the design provides a **holistic view** of the system, closely aligned with system objectives and requirements, but as iterations continue, the design becomes more specific and detailed, addressing individual data, functional, and behavioral components.

Key Principles of Software Design

1. **Iteration and Refinement**: Software design is not a linear process. Initial designs are abstract and general, but through subsequent iterations, these designs are refined and detailed to accommodate all system requirements.
2. **Blueprint Creation**: The design process ultimately results in a blueprint that software developers use to guide implementation. The design process ensures that each level of abstraction stays aligned with the project's requirements.

Software Quality Guidelines and Attributes

The quality of a software design is assessed through **technical reviews** and must adhere to certain **guidelines** and **quality attributes**.

Quality Guidelines

These guidelines help maintain high design standards and ensure that the design is effective and implementable:

1. **Architectural Foundation**: The design must incorporate recognizable **architectural styles or patterns** and include well-structured components. It should allow for **evolutionary implementation** (e.g., facilitating easier testing and implementation).
2. **Modularity**: Software should be logically divided into subsystems or components that handle specific responsibilities.
3. **Clear Representation**: The design should clearly represent **data**, **architecture**, **interfaces**, and **components**.
4. **Data Structures**: The design should suggest appropriate data structures, ensuring that these structures are practical and efficient for the intended classes.
5. **Independent Components**: The system's components should have **independent functionality**, making them easier to maintain and test.
6. **Simplified Interfaces**: Interfaces should reduce the complexity of interactions between different components and between the system and the outside world.
7. **Repeatable Design Process**: The design should be **repeatable** and driven by the **requirements analysis**, ensuring consistency across design iterations.
8. **Effective Communication**: The design must be documented in a way that communicates its purpose clearly to the developers who will build and maintain the system.

Quality Attributes (FURPS)

Hewlett-Packard's **FURPS model** defines five quality attributes that serve as targets for software design. These attributes represent various aspects of software that need to be considered to build a high-quality system:

1. **Functionality**: Focuses on the **feature set**, capabilities, and security of the system. The goal is to ensure the system does what it's supposed to do.
2. **Usability**: Concerns human factors, user interface design, **aesthetics**, consistency, and documentation. It's about making the system easy to use and understand.
3. **Reliability**: Measured by **failure rates**, **accuracy of output**, **mean-time-to-failure (MTTF)**, **recovery mechanisms**, and **predictability**. A reliable system is stable, accurate, and can recover from failures.
4. **Performance**: Concerns **speed**, **response time**, **resource consumption**, and **throughput**. Performance is important for systems with high user demands or time-sensitive operations.
5. **Supportability**: Encompasses **maintainability** (extensibility, adaptability, serviceability), **testability**, **compatibility**, **configurability**, and ease of installation. Supportability ensures that the system can evolve, be tested, and be easily fixed or extended in the future.

The **weighting** of these quality attributes will vary depending on the project's needs. For example, a financial system may prioritize **functionality** and **reliability**, while a media streaming app may focus more on **performance** and **usability**.

The Evolution of Software Design

Software design has **evolved significantly** over the last several decades, shaped by various methodologies and paradigms:

1. **Modular Design**: Early efforts focused on creating **modular programs** and refining software structures using **top-down** approaches.
2. **Structured Programming**: The development of **structured programming** emphasized procedural design, focusing on controlling complexity with clear structures and flow control.
3. **Data Flow and Data Structures**: Later, methods for designing systems based on **data flow** or **data structures** emerged. These helped better define system components in terms of data.
4. **Object-Oriented Design (OOD)**: The introduction of **object-oriented design** focused on defining objects with attributes and behaviors, leading to better abstraction and reuse.
5. **Software Architecture and Design Patterns**: More recent focus has been on **software architecture** and **design patterns**. Architectural styles and patterns provide templates for designing robust systems at a higher level of abstraction.
6. **Aspect-Oriented, Model-Driven, and Test-Driven Development**: These newer approaches focus on achieving modularity and clear separation of concerns. For example:
    - **Aspect-Oriented Programming** helps separate cross-cutting concerns (e.g., logging, security).
    - **Model-Driven Development** emphasizes automated code generation from models.
    - **Test-Driven Development (TDD)** focuses on building software iteratively with tests written before code.

Common Characteristics of Software Design Methods

No matter the design methodology chosen, there are several **common characteristics** shared by most software design approaches:

1. **Translation of Requirements**: A process for transforming the system's requirements into a design.
2. **Notation for Functional Components**: A notation for clearly representing the components of the system and how they interact.
3. **Heuristics for Refinement**: Methods for refining the design, including partitioning the system into smaller, manageable parts.
4. **Guidelines for Quality Assessment**: Rules and standards to evaluate the design's effectiveness and alignment with quality attributes.

The **software design process** is a continuous and iterative effort to create a well-structured blueprint for the system. Throughout this process, attention must be paid to key **quality attributes** (FURPS) and **guidelines** that lead to a robust, maintainable, and performant system. The evolution of design practices, from **modular programming** to **object-oriented** and **architecture-focused** approaches, has greatly enhanced the capability of software teams to create complex systems that meet various stakeholder needs. Regardless of the specific design method, applying sound design principles, systematic methods, and frequent reviews is key to achieving a successful software product.

2.7 Design concept

The text you've provided is a detailed discussion on fundamental software design concepts, ranging from abstraction to object-oriented design. Here's a brief summary of each key concept:

1. Abstraction

- Abstraction involves simplifying complex systems by focusing on the essential aspects while hiding the irrelevant details. This can be broken down into:
  - **Procedural Abstraction**: Describes a sequence of operations (e.g., opening a door).
  - **Data Abstraction**: Represents data objects and their attributes (e.g., a door's properties like type, weight).

2. Architecture

- Software architecture is the high-level structure of a system, including the components and their interactions. Architectural patterns help address common design problems. It includes:
  - **Structural Properties**: Defines how components interact.
  - **Extra-functional Properties**: Addresses system characteristics like performance and reliability.
  - **Families of Related Systems**: Reusable patterns for similar systems.

3. Patterns

- Design patterns offer proven solutions to recurring problems in a specific context. They help software designers reuse solutions, saving time and guiding design.

4. Separation of Concerns

- This concept suggests dividing a complex problem into smaller, independent sections. This makes the problem easier to solve and manage. It is often used to design modular software systems.

5. Modularity

- Modularity is the division of software into manageable components (modules). Proper modularization leads to software that is easier to understand, test, maintain, and integrate.

6. Information Hiding

- Information hiding suggests that modules should only expose what's necessary for others to interact with them, concealing implementation details to prevent unwanted dependencies and improve maintainability.

7. Functional Independence

- Modules should be designed to perform specific tasks independently, with minimal interaction between them. This simplifies development, testing, and maintenance.

8. Refinement

- Stepwise refinement is the process of breaking down complex tasks into simpler components, progressing from abstract ideas to detailed implementations.

9. Aspects

- Aspects are cross-cutting concerns that affect multiple modules or components in a system. The design should isolate these concerns into separate modules to avoid scattering them throughout the system.

10. Refactoring

- Refactoring involves restructuring existing software to improve its internal design without altering its external behavior. It's used to eliminate inefficiencies, improve code quality, and maintainability.

11. Object-Oriented Design Concepts

- Object-Oriented Design (OOD) uses concepts like classes, inheritance, and polymorphism to model real-world entities and their interactions within the software.

12. Design Classes

- Design classes are refined from analysis classes and serve as the building blocks of the system's implementation. These include:
  - **User Interface Classes**: Handle human-computer interaction.
  - **Business Domain Classes**: Represent key business logic.
  - **Process Classes**: Manage lower-level business abstractions.
  - **Persistent Classes**: Manage data storage.
  - **System Classes**: Handle system-level functions.

These concepts aim to guide software engineers in creating maintainable, flexible, and efficient software designs. By focusing on modularity, abstraction, and separation of concerns, software systems become easier to manage, scale, and modify over time.

2.8 DESIGN MODEL DIMENSIONS

Design modeling in software engineering encompasses the detailed creation of software structures, components, and interfaces. The design model serves as the blueprint for developing the final system and includes various dimensions that evolve over time. Here's an overview of the key design model dimensions and elements:

1. The Design Model Dimensions

Process Dimension

The **Process Dimension** refers to the sequence of design tasks that occur throughout the software development lifecycle. It captures how design activities unfold and progress, emphasizing the continuous, iterative nature of software design. This dimension includes:

- **High-Level Design**: Focuses on the system's overall structure and major components.
- **Detailed Design**: Moves from abstract concepts to specific technical implementations (e.g., algorithms, data structures, hardware requirements).

Design tasks are iterative, with multiple stages of refinement, revisiting earlier models based on new information, feedback, and evolving requirements. For example, an initial architectural design may need to be revisited after data design and component-level decisions are made.

Abstraction Dimension

The **Abstraction Dimension** outlines the progression from high-level abstract models to more detailed, implementation-specific designs. This dimension helps manage complexity by breaking down a system into manageable layers of abstraction.

- **High-Level Models**: Often conceptual or focused on major system functionalities and interactions (e.g., data flow diagrams, use case diagrams).
- **Low-Level Models**: More detailed and close to the implementation phase, including specifics like data structures, algorithms, and component behaviors.

In the design phase, the boundary between **analysis** and **design** can become blurred, as elements of the analysis model directly evolve into design artifacts, reducing rework and improving continuity.

2. Data Design Elements

Data design is a key aspect of the overall system design, as it impacts the internal data structures, database systems, and business-level data systems. The **Data Design Element** involves translating a high-level data model into more detailed forms suited for implementation.

Key Aspects of Data Design:

- **Data Structures**: Involves defining how data is organized and manipulated within the system (e.g., arrays, linked lists, hash tables, databases).
- **Database Models**: If the system involves large-scale data storage, it will likely require a database schema, which can include relational models, NoSQL databases, or other specialized data stores.
- **Business-Level Data**: Involves the design of systems like **data warehouses** that support business analytics and decision-making processes.

Data design often occurs iteratively, starting from an abstract data model (like an Entity-Relationship diagram) and evolving into more detailed database schemas and storage mechanisms. The goal is to create an efficient, maintainable data structure that supports the system's needs.

3. Architectural Design Elements

Architectural design is one of the most important aspects of system design, defining the structure and major components of the software. It provides a high-level view of the software's organization, much like a house's floor plan.

Key Aspects of Architectural Design:

- **Subsystems**: The system is broken down into smaller, manageable subsystems. Each subsystem focuses on a particular function of the software (e.g., user interface, data processing, authentication).
- **Design Patterns**: Patterns such as **MVC (Model-View-Controller)** or **Client-Server** may be applied to organize components and their interactions.
- **Interrelationships**: Defines how different components and subsystems interact and communicate with each other.

The architecture is derived from several sources:

- **Application Domain Knowledge**: Understanding the problem domain helps guide architectural decisions (e.g., for a real-time system, the architecture might emphasize responsiveness and fault tolerance).
- **Requirements**: The system's functional and non-functional requirements dictate how the architecture is structured (e.g., scalability, security, and performance considerations).
- **Existing Architectural Styles**: Leveraging established patterns (e.g., microservices, layered architecture) helps speed up the design process and improve maintainability.

Architectural design usually leads to the creation of **architectural models** (e.g., UML component diagrams or deployment diagrams).

4. Interface Design Elements

Interface design defines how the software communicates with its users, other systems, or internal components. It is crucial for ensuring effective communication between the software and its environment.

Types of Interfaces:

- **User Interface (UI)**: Focuses on how users interact with the system. It includes visual elements (e.g., buttons, forms) and the flow of user actions.
- **External Interfaces**: Define how the system communicates with other systems, devices, or networks. These may involve APIs, protocols, or data exchange formats (e.g., RESTful APIs, WebSockets).
- **Internal Interfaces**: Concern the communication between different internal components of the software system (e.g., communication between modules, database interactions).

Good interface design ensures that information flows smoothly and intuitively between the system and users, while maintaining efficiency and reliability when interacting with external systems.

5. Component-Level Design Elements

Component-level design delves into the detailed design of individual software components. It is akin to developing the detailed plans for rooms in a house, specifying the internal structure and behavior.

Key Aspects of Component-Level Design:

- **Data Structures**: Defines the internal data management, such as how data is stored, accessed, and manipulated within each component (e.g., linked lists, trees).

- **Algorithms**: Specifies the logic and steps to be followed for tasks within the component (e.g., search algorithms, sorting algorithms, encryption).
- **Component Behavior**: Describes how the component will behave when triggered by events or messages.

At this level, detailed specifications are created for each component, including **pseudocode**, **UML activity diagrams**, or even **flowcharts** to represent the logic flow.

6. Deployment-Level Design Elements

Deployment design focuses on the **physical deployment** of the system, mapping how software components will be distributed across physical hardware, devices, and networks.

Key Aspects of Deployment Design:

- **Deployment Diagram**: A UML deployment diagram shows the distribution of software subsystems across various hardware devices or servers, helping visualize how the software interacts with its environment.
- **Hardware Configurations**: Specifies which devices, operating systems, and other hardware configurations will be used to run the system (e.g., Mac, Windows, Linux).
- **Distributed Systems**: Involves considerations for deploying distributed systems, where components are spread across different machines or geographical locations.

Deployment-level design ensures that the system will be able to operate in the specified physical environments and meet performance and scalability requirements.

Key Concepts and Tools in Design Modeling

- **UML Diagrams**: UML (Unified Modeling Language) plays a key role in software design. Various types of diagrams such as **class diagrams**, **sequence diagrams**, **component diagrams**, and **deployment diagrams** are used to represent different aspects of the system at various stages.
- **Design Patterns**: Reusable solutions to common design problems (e.g., Singleton, Factory, Observer) are crucial at different stages of design. Applying design patterns ensures the software is both efficient and maintainable.

Overall Design Process

The software design process is **iterative** and **evolutionary**, with each element of the design being refined as the development process advances. Some design tasks occur sequentially (e.g., architectural design followed by component design), while others, especially lower-level tasks, can be done concurrently. The design model is continuously improved based on new requirements, system feedback, and testing results.

The design model in software engineering involves several dimensions and elements that together form a comprehensive blueprint for building the software. From **high-level architecture** to **low-level components**, each dimension plays a crucial role in guiding the system from abstract concepts to detailed implementation. By understanding and applying design principles, patterns, and tools, engineers can create well-structured, maintainable, and scalable software systems.

2.9 SOFTWARE ARCHITECTURE

This section delves into the concept of software architecture, offering a detailed exploration of its importance, definitions, components, and how it is represented. Below is a summary and breakdown of the key points:

1. What is Architecture?

- **Building Architecture**: In the context of buildings, architecture is not just about the shape of the structure but how the components (rooms, materials, etc.) are integrated to create a functional and aesthetically pleasing whole. It includes thousands of design decisions, large and small, made at different stages of the process.
- **Software Architecture**: Similarly, software architecture involves making design decisions early in the process that will shape the structure of the system. According to Bass, Clements, and Kazman, software architecture is the **structure of the system**, comprising components, their externally visible properties, and the relationships between them. These decisions, once made, are often difficult to change later and influence every aspect of the software's design and functionality.

2. Why Is Architecture Important?

- **Communication**: Software architecture serves as a tool for communication among stakeholders (e.g., developers, customers, designers) by providing a common language and framework for discussing the system's structure.
- **Early Decisions**: Architecture identifies crucial early design decisions that can have long-term impacts on the system’s success, influencing everything from functionality to performance.
- **Graspable Model**: It offers a manageable abstraction of the system's design, which allows engineers to understand the system's structure and how its components interact, making the development process more predictable.

3. Architectural Descriptions

- **Multiple Views**: Just as in architecture for buildings, a software system’s architecture needs to be viewed from multiple perspectives to address different stakeholder concerns. For example:
    - The **owner** of a building might care about aesthetics and functionality.
    - The **structural engineer** needs detailed plans of the building’s framework.

- Similarly, in software, the system must be described using various **views** (representations of the whole system from specific perspectives), with each view addressing a particular concern (e.g., performance, security, scalability).
- **IEEE-Std-1471-2000** defines the **Architectural Description (AD)** as a collection of work products that represent the architecture from different views, and outlines how these should be documented.

4. Architectural Decisions

- **Decisions and Rationale**: Each architectural choice—be it related to the system's structure, components, or interactions—should be carefully documented to capture the reasoning behind it. These decisions form part of the architecture's documentation and provide context for future design changes.
- **Template for Decision Documentation**: A system architect should document the reasoning for significant architectural decisions. This historical record helps future architects and designers understand the evolution of the architecture and aids in making informed decisions when modifications are necessary.

Key Concepts:

- **Stakeholder Concerns**: Different stakeholders (customers, developers, engineers) have different concerns that need to be addressed through tailored views of the architecture.
- **Architecture vs. Design**: Architecture is often seen as a higher-level, conceptual framework, while design involves the specific instantiations or implementations of that architecture in the system.
- **Reusable Architectural Patterns**: Over time, patterns emerge in software design that can be reused across different systems. These patterns are a core part of the architectural language that helps manage complexity and ensure scalability, reliability, and maintainability.

Overall Structure:

- **Early Design Decisions**: These are crucial in shaping the overall success of the system.
- **Representations and Views**: These help visualize and communicate different aspects of the architecture, ensuring that all stakeholders understand how the system will function.
- **Architectural Documentation**: A record of decisions made throughout the architectural design process helps ensure clarity and offers guidance for future revisions.

2.10 ARCHITECTURAL STYLES

This section outlines various **architectural styles** and their relevance in the design of software systems. Just as in building architecture where a style defines the overall design and functionality (like "center hall colonial"), software architecture also involves certain styles that guide the overall structure and design of a system. Let's break down the key concepts:

1. Architectural Styles in Software

An **architectural style** in software refers to a pattern or structure that defines how the components of a system are organized and how they interact. The idea is that certain patterns are suited to specific types of problems or domains, much like a "style" of a building suggests specific features and layouts.

Key Features of Architectural Styles**:**

- **Components**: These are the individual building blocks of the system (e.g., databases, computational modules).
- **Connectors**: These allow communication, coordination, and cooperation between the components (e.g., data channels, protocols).
- **Constraints**: These define how components should be integrated to form the system, ensuring the system works as a cohesive whole.
- **Semantic Models**: These models allow designers to understand the properties of the system by analyzing the components' known characteristics.

Differences Between Architectural Styles and Patterns**:**

- **Architectural Styles**: Impose a transformation on the design of the entire system and define a broad structure.
- **Architectural Patterns**: Focus on solving specific issues (like concurrency or real-time behavior) within the architecture, addressing particular aspects of the system's behavior rather than its overall structure.

2. Common Architectural Styles

These are some of the most widely recognized styles in software architecture:

Data-Centered Architecture:

- **Concept**: A central data store (like a database or file) is accessed by various components, which can modify or retrieve the data.
- **Use Case**: Common in systems like database-driven applications.
- **Example**: A **blackboard architecture** is a variation where the central repository notifies components about data changes.
- **Advantages**: Easy to add new components (clients) that independently interact with the data store, promoting **integrability**.

Data-Flow Architecture:

- **Concept**: Data flows through a series of transformations, typically represented as components called filters.
- **Example**: The **pipe-and-filter** pattern, where each filter processes the data and passes it on to the next filter. Each filter operates independently of others.
- **Use Case**: Often used in systems like compilers, data processing pipelines, or any system that processes data step-by-step.
- **Advantages**: Modular and scalable, as each component can work independently.

Call and Return Architectures:

- **Concept**: This architecture is based on a hierarchical structure, where a main program calls subprograms (or functions) that, in turn, may call other subprograms.
- **Substyles**:
  - **Main Program/Subprogram**: A simple structure where a control hierarchy exists, and the main program invokes various components.
  - **Remote Procedure Call (RPC)**: Extends this concept to distributed systems, where the subprograms reside on different computers connected via a network.
- **Advantages**: Easy to understand and modify, scalable for distributed systems.

Object-Oriented Architectures:

- **Concept**: The system is built around objects that encapsulate both data and behavior. Components communicate through message passing.
- **Use Case**: Suitable for systems where objects need to interact dynamically (like GUI-based systems, games, or large-scale enterprise applications).
- **Advantages**: Promotes reuse and encapsulation, making the system easier to maintain.

Layered Architectures:

- **Concept**: The system is divided into layers, each performing a different set of functions. The outermost layer interacts with the user, and inner layers handle the operating system, utilities, etc.
- **Example**: A common example is the **OSI model** used in networking, where layers include physical, data link, network, transport, etc.
- **Use Case**: Often used in systems that require clear separation of concerns and modularity, such as enterprise software and web applications.
- **Advantages**: Separation of concerns leads to better maintainability and scalability.

3. Choosing an Architectural Style

When designing a system, the **requirements engineering** phase helps identify the system's characteristics, constraints, and domain, which guide the selection of an appropriate architectural style. Often, more than one architectural style can be used in combination to fit the specific needs of the application.

- For instance, in a **database-driven web application**, a **layered architecture** might be used for overall structure, combined with a **data-centered architecture** for managing the database and interactions between the layers.

4. Architectural Patterns

Architectural patterns are used to solve recurring problems in software design. They address specific issues such as concurrency, real-time communication, or the handling of large amounts of data in particular contexts.

- **Example**: An **e-commerce platform** might need to handle different types of customers and products. The pattern could suggest specific ways of structuring product catalogs, customer databases, and payment systems.
- **Purpose**: Patterns are tailored solutions to recurring problems and can be used within an overall architectural style to address domain-specific needs.

5. Key Design Considerations

To evaluate which architectural style fits best for a given system, designers should consider several factors:

- **Control**: How is control managed within the system? Is it hierarchical or distributed? How are components synchronized?
- **Data**: How does data flow between components? Is it shared globally or passed on-demand? Are data components passive or active?

In summary, **architectural styles** define the overall structure and organization of a system, while **architectural patterns** address specific functional or behavioral concerns. Choosing the right style—or a combination of styles—requires a deep understanding of the system's needs, and these decisions ultimately shape how the software will behave, scale, and evolve.

2.11 ARCHITECTURAL MAPPING USING DATA FLOW

The **mapping process** that translates a **requirements model** into a **software architecture** using different **architectural styles**. It also emphasizes the importance of **refining architectural designs** to meet both functional and performance requirements.

Mapping Requirements to Architectural Styles

The architectural styles discussed earlier (like call-and-return, data-centered, and layered architectures) do not always have a straightforward mapping from the requirements model. For some styles, there is no direct mapping, and the designer must apply different techniques to map the requirements to the architectural design.

A key mapping technique mentioned here is **structured design**, which is used to transition from data flow diagrams (DFDs) to an architectural structure. This method is often used when the system exhibits **data flow-oriented** characteristics, where information flows through various processes or components.

Structured Design Process (for Data Flow-Oriented Systems)

In the case of a **call-and-return architecture** (which is a common architectural style), it can fit within more sophisticated architectures, such as **client-server** systems. The structured design method provides a way to create a mapping between the information flow in a DFD and a program structure. Here's an outline of the steps involved in this mapping process:

1. **Review the System Model**: Understand the fundamental system model, which in the example involves the security function of a system like SafeHome.
2. **Refine DFDs**: Refine the DFDs to include more details, breaking down the processes into smaller components. These will help create more accurate architectural components.
3. **Identify Transform or Transaction Flow**: The next step is to evaluate the DFDs for the type of information flow. A **transform flow** suggests linear transformations, where data is processed and converted from one form to another.
4. **Define Flow Boundaries**: Set boundaries for incoming and outgoing data flows to clearly define where the data is processed. The boundaries will help structure the architecture and identify which components will handle the data transformation.
5. **First-Level Factoring**: This step involves creating a top-down distribution of control, defining high-level components (controllers) that manage incoming data, processing, and outgoing data.
6. **Second-Level Factoring**: The second-level involves mapping individual transforms in the DFD to specific modules within the architecture. It provides more detailed control over how the data is processed within each component.
7. **Refine the Architecture**: After establishing the initial structure, it's important to refine the architecture by applying design heuristics to improve cohesion, reduce coupling, and ensure the system can be efficiently implemented and maintained.

Refining the Architectural Design

The refinement of the architecture aims to optimize it for **modularity**, **functional independence**, and **maintainability**. The passage notes that achieving an "optimal design" is not necessarily the goal if the design cannot meet functional and performance requirements. It also mentions that **design measures and heuristics** should guide the refinement process.

Key goals in this refinement process include:

- **Simplicity**: The refined architecture should maintain a **simple structure** with as few components as necessary while ensuring effective modularity.
- **Elegance and Efficiency**: The simplest possible architecture that meets requirements should be pursued. This often results in a design that is both elegant and efficient.

Practical Example: SafeHome Security System

The **SafeHome security system** is used as an example of the transformation of DFDs into software architecture. The security system's data flow diagrams (DFDs) are mapped into a **call-and-return architecture**, with control distributed across several modules:

- **Monitor Sensors Executive**: The main controller of the system.
- **Sensor Input Controller**: Responsible for managing incoming data from sensors.
- **Alarm Conditions Controller**: Processes internalized data.
- **Alarm Output Controller**: Handles the outgoing information, such as triggering alarms or sending notifications.

The goal is to refine these high-level components into smaller, more specific modules that will carry out distinct functions.

Key Takeaways

- **Mapping** from requirements to architecture can be complex, and there may be no direct translation for some architectural styles.
- The **structured design method** is a useful technique for mapping **data flow diagrams** (DFDs) into specific **architectural styles**, such as call-and-return.
- The process involves steps like reviewing system models, defining flow boundaries, and factoring control into layers of components.
- **Refinement** is crucial for improving the architecture to make it simple, modular, and easy to maintain. It involves minimizing complexity while maintaining the functionality required by the system.

2.12 UI Analysis and Design Models

An overview of the **user interface (UI) analysis and design process**. It highlights the importance of understanding the users, their tasks, and the system's functionality in order to create an interface that is both effective and easy to use. The process is described as **iterative**, using a **spiral model** that allows for continuous refinement based on feedback and testing.

UI Analysis and Design Models

Four key models are involved in UI analysis and design:

1. **User Model**: This represents the profile of the system's end users. It includes demographic information (age, gender, cultural background) as well as cognitive factors (motivation, goals, skill level). The user model helps the designer understand

the needs of different user groups (novices, intermittent users, and frequent users) and the specific interaction challenges they might face.

2. **Design Model**: The design model is created by the software engineer and represents how the interface is structured and what actions and objects are needed to facilitate the users' tasks. The design model should be aligned with the user model, ensuring it accommodates the specific needs of users, both in terms of **syntax** (how users interact with the interface) and **semantics** (the meaning and purpose of these interactions).
3. **Mental Model**: This is the user's **mental representation** of how the system works, which may not be entirely accurate but guides their interaction with the system. A well-designed UI should aim to align the system's behavior with the user's mental model to make it intuitive.
4. **Implementation Model**: This model refers to the physical manifestation of the interface, including visual elements and supporting materials (manuals, help files, etc.). The implementation model must reflect the syntactic and semantic information necessary for the user to understand and interact with the system effectively.

The Process of UI Analysis and Design

The process is **iterative**, meaning it involves revisiting the activities multiple times to refine and enhance the design. The four main activities in the spiral process are:

1. **Interface Analysis and Modeling**: This involves gathering information about the users, their tasks, and the environment in which they will use the system. This phase includes:
    - **User profile analysis**: Understanding user skill levels, goals, and other characteristics.
    - **Task analysis**: Identifying the tasks users will perform to achieve their goals.
    - **Environment analysis**: Evaluating physical conditions (e.g., space, noise) that could affect how the interface is used.
2. **Interface Design**: Once the analysis is complete, the interface design begins. The aim is to define **interface objects and actions** (e.g., buttons, forms, navigation) and ensure that these are represented effectively on the screen. The design should meet **usability goals** and allow users to complete their tasks intuitively.
3. **Interface Construction**: The prototype is built during this phase, allowing for real-world testing of the interface. This iterative process helps ensure that the design is practical and meets user expectations. A **UI toolkit** can help speed up the construction of the interface by providing pre-made components and tools.
4. **Interface Validation**: This phase ensures that the interface meets all functional and usability requirements. Validation focuses on:
    - Correctness: Ensuring the interface performs all tasks as required.
    - Usability: Ensuring the interface is easy to use and learn.
    - Acceptance: Ensuring that users find the interface useful and accept it in their work environment.

Iterative Nature of the Process

- Each of the phases mentioned is not a one-time activity but is revisited multiple times to refine and improve the UI design.
- Early iterations focus on **general requirements** and **high-level designs**, while later iterations focus on **more specific tasks** and **detailed interface components**.

- The process involves **prototyping** to gather feedback from users early and often, ensuring that the design evolves based on real user input.

Key Points from This Section:

- **Know the user**: The design of the interface should be based on a deep understanding of the users' goals, tasks, and characteristics. This includes gathering detailed user profiles and task information.
- **Task and environment analysis**: The tasks users perform and the environment in which they work are key considerations in creating a useful interface.
- **Iterative design**: Interface design is not a linear process but an iterative one, with multiple passes around the spiral to refine the system as new requirements emerge.
- **Prototyping**: Prototypes are essential for testing and validating the design. By simulating the interface early in the process, developers can receive feedback and make adjustments before full implementation.

In summary, the goal of UI analysis and design is to create an interface that is both effective and user-friendly by thoroughly understanding the users, their tasks, and the environment in which they work. The process is iterative and requires constant testing and refinement to ensure the interface meets the needs of its users.