UNIT III Design Engineering and Creating an Architectural Design

Design Engineering

Design Process and Design Quality

Design Concepts

The Design Model

Creating an Architectural Design

Software Architecture

What Is Architecture?

Architectural design in software engineering encompasses the organizational structure of data and program components necessary for constructing a computer-based system. It involves:

1. Architectural Style: Determining the overall style or pattern that the system will adhere to.
2. Component Structure: Defining the structure and characteristics of the system's components.
3. Interrelationships: Identifying and understanding the connections and interactions among all architectural components.

Architecture serves as a blueprint that allows software engineers to:

- Analyze Effectiveness: Evaluate how well the design meets the specified requirements.
- Consider Alternatives: Explore different architectural options while it's still feasible to make changes.
- Reduce Risks: Mitigate risks associated with software construction by addressing architectural concerns early in the process.

Software architecture design involves two primary levels:

1. Data Design: Concerned with representing the data component of the architecture.

2. Architectural Design: Focuses on defining the structure of software components, their attributes, and how they interact with each other.

Data Design

Data design is a crucial activity in software engineering that involves translating data objects from the analysis model into data structures at the software component level. This process also includes establishing a database architecture at the application level, if required. Here are key aspects of data design at different levels:

At the Program Component Level:

- Designing data structures and algorithms necessary for manipulating them is vital for creating high-quality applications.
- Well-designed data structures and algorithms contribute to the efficiency and effectiveness of software components.

At the Application Level:

- Translating a data model derived from requirements engineering into a database is essential for achieving the business objectives of a system.
- The collection and organization of information stored in databases into a "data warehouse" facilitate data mining or knowledge discovery, which can significantly impact business success.

Data Design at the Architectural Level:

- Businesses face challenges in extracting useful information from diverse data environments, especially when information is needed across functional areas.
- Techniques like data mining (or knowledge discovery in databases) and data warehouses address these challenges by navigating existing databases and organizing relevant information for business-level insights.

Data Design at the Component Level:

- Focuses on representing data structures directly accessed by software components.
- Principles for data specification include systematic analysis, identification of data structures and operations, establishment of a data dictionary, deferral of low-level design decisions, controlled access to data structures, development of a data structure library, and support for abstract data types in programming languages.

Architectural Styles and Patterns

Architectural styles and patterns play a crucial role in designing and organizing software systems. They provide a structured approach to differentiate and categorize different types of systems, each with its own set of components, connectors, constraints, and semantic models. Here are some key points about architectural styles and patterns:

Architectural Styles:

- An architectural style describes the overall structure of a software system and how its components and connectors interact.
- Each architectural style encompasses a set of components performing specific functions, connectors facilitating communication among components, integration constraints, and semantic models for understanding system properties.
- Examples of architectural styles include data-centered architectures, data-flow architectures, and call-and-return architectures.

Architectural Patterns:

- Architectural patterns are specific solutions or templates that impose design rules on the architecture to handle particular aspects of functionality.
- Unlike architectural styles, patterns focus on specific behavioral issues within the context of the architecture.
- Patterns impose rules at the infrastructure level, defining how software components handle certain functionalities.
- Architectural patterns are less broad in scope compared to architectural styles and tend to address specific design concerns.

Taxonomy of Styles and Patterns:

1. **Data-Centered Architectures:**
   - Data store (e.g., a file or database) is at the center and accessed by other components.
   - Promotes integrability and facilitates data exchange using mechanisms like the blackboard pattern.
2. **Data-Flow Architectures:**
   - Input data are transformed through a series of computational components into output data.
   - Follows patterns like pipe and filter, which process data independently and in a specified form.
3. **Call and Return Architectures:**
   - Enables easy modification and scalability of program structure.

- Includes substyles like main program/subprogram, remote procedure call, object-oriented, and layered architectures.

Architectural Patterns:

- Architectural patterns provide specific solutions for handling behavioral aspects of a software system.
- They differ from architectural styles by having a narrower scope, focusing on specific aspects rather than the entire architecture.
- Patterns impose rules on the infrastructure level, describing how software should handle certain functionalities.
- Address specific behavioral issues within the context of the architectural design.

Common Architectural Patterns:

1. **Concurrency Patterns:**
   - Address the need to handle multiple tasks simultaneously.
   - Examples include the operating system process management pattern and task scheduler pattern.
2. **Persistence Patterns:**
   - Deal with data persistence beyond the execution of the creating process.
   - Common patterns include the database management system pattern and application-level persistence pattern.
3. **Distribution Patterns:**
   - Focus on communication between systems or components in a distributed environment.
   - Examples include the broker pattern, where a broker acts as a middleman between client and server components.

Organization and Refinement:

- Design processes often result in multiple architectural alternatives, requiring the establishment of design criteria for assessment.
- Key questions to gain insight into the derived architectural style include:
  - Control: How is control managed? Is there a distinct hierarchy, and how is control shared?
  - Data: How is data communicated? Is the flow continuous, and what is the mode of transfer? Are data components present, and how do they interact?
  - Interaction: How do functional components interact with data components? Are data components passive or active?

Architectural Design

Representing the System in Context:

- At the architectural design level, an architectural context diagram (ACD) models how software interacts with external entities.
- Superordinate systems use the target system as part of higher-level processing, while subordinate systems provide data or processing.
- Actors interact with the target system by producing or consuming necessary information.

Defining Archetypes:

- Archetypes represent core abstractions critical to the design.
- Derived from analysis classes, archetypes for SafeHome security function include Node, Detector, Indicator, and Controller.

Refining the Architecture into Components:

- The architectural designer begins with analysis model classes, addressing entities within the application domain.
- Infrastructure components, like memory and communication management, are integrated into the architecture.
- Top-level components for SafeHome security function include External Communication Management, Control Panel Processing, Detector Management, and Alarm Processing.

Describing Instantiations of the System:

- Actual instantiation applies the architecture to a specific problem to demonstrate its appropriateness.

Object and Object Classes:

- An object has a state and defined operations.
- Object class definition includes attributes and operations associated with objects of that class.

Object-Oriented Design Process:

1. Understand and define the system's context and modes of use.
2. Design the system architecture.
3. Identify principle objects.

4. Develop design models.
5. Specify object interfaces.

Systems Context and Modes of Use:

- Describes the system's relationships with its external environment, either static or dynamic.

System Architecture:

- Based on the defined interaction between the software system and its environment.

Object Identification:

- Identify object classes using grammatical, tangible entities, behavioral, or scenario-based approaches.

Design Models:

- Static models describe relationships between objects, while dynamic models describe interactions.

Object Interface Specification:

- Specifies details of object interfaces.

Design Evolution:

- Object-oriented design simplifies making changes without affecting other system objects' functionality.

Conceptual Model of UML

CONCEPTUAL MODEL OF UML

A conceptual model serves as a foundation for understanding and constructing UML diagrams, depicting real-world entities and their interactions. Mastering the conceptual model of UML involves understanding three major elements:

1. UML Building Blocks:
    - These are the fundamental elements used to construct UML diagrams.
    - Building blocks include classes, objects, interfaces, relationships, and various types of diagrams like class diagrams, sequence diagrams, and use case diagrams.

2. **Rules to Connect the Building Blocks:**
    - Once the building blocks are identified, it's essential to understand how they connect and interact with each other.
    - Rules govern the relationships between building blocks, such as associations, dependencies, generalizations, and realizations.
3. **Common Mechanisms of UML:**
    - UML incorporates several common mechanisms to enhance modeling and communication:
        - Encapsulation: Protecting the internal state of objects.
        - Inheritance: Establishing relationships between classes.
        - Polymorphism: Allowing objects to take on multiple forms.
        - Abstraction: Focusing on essential characteristics while hiding irrelevant details.
        - Modularity: Breaking down systems into manageable components.

Basic Structural Modeling

The Unified Modeling Language (UML) provides a standardized way to visualize the design of software systems, including their structure, behavior, and interactions. In UML, structural modeling focuses on representing the static structure of a system, including its components, relationships, and constraints. Here's an overview of the basic structural modeling concepts in UML:

1. **Class:** Blueprint for objects.
2. **Object:** Instance of a class.
3. **Attribute:** Properties of a class.
4. **Method:** Behavior of a class.
5. **Association:** Relationship between classes.
6. **Generalization (Inheritance):** "Is-a" relationship.
7. **Aggregation:** "Whole-part" relationship.
8. **Composition:** Stronger form of aggregation.
9. **Interface:** Contract for implementing classes.
10. **Package:** Grouping mechanism for elements.
11. **Dependency:** Relationship indicating one element depends on another.

Class Diagrams

A class diagram is a type of diagram in the Unified Modeling Language (UML) that represents the structure of a system by showing the system's classes, their attributes,

methods, and the relationships among objects. Here's a brief overview of some of the key elements you mentioned:

1. Classes: Classes represent objects in the system and define their attributes (data fields) and methods (functions or procedures). Each class typically has a name, attributes, and methods.
2. Packages: Packages provide a way to organize classes into groups or modules. They help in managing the complexity of large systems by providing a hierarchical structure.
3. Objects: Objects are instances of classes. They represent individual entities within the system and hold values for the attributes defined by their class.
4. Containment: Containment relationships depict how classes or objects may contain or own other classes or objects. For example, a university might contain departments, which in turn contain professors and students.
5. Inheritance: Inheritance represents an "is-a" relationship between classes, where one class (subclass or child class) inherits attributes and methods from another class (superclass or parent class). It allows for code reuse and supports the concept of polymorphism.
6. Associations: Associations represent relationships between classes or objects. They can be uni-directional or bi-directional and can have multiplicities indicating the number of objects involved in the relationship.

Sequence Diagrams

Sequence Diagram:

- A sequence diagram in UML is used to represent the interactions between objects or components in a system over time.
- The vertical dimension typically represents time, with events and interactions occurring from top to bottom.
- The horizontal dimension represents different objects or components involved in the interaction.
- Objects are shown as vertical lines (known as lifelines) across the top of the diagram.
- Messages between objects are represented by arrows or lines, indicating the flow of communication.
- Sequence diagrams are particularly useful for understanding the flow of control and the sequence of messages exchanged between objects during a particular scenario or use case.

Collaboration Diagrams

Collaboration diagrams, also known as Communication Diagrams, illustrate how objects or components interact with each other to achieve specific functionalities within a system. Here's a succinct breakdown:

- Objects/Components: Represent entities or modules within the system.
- Links/Associations: Show connections and interactions between objects/components.
- Messages: Indicate communication between objects/components.
- Sequence Numbers: Provide the order of message exchanges.
- Structural Organization: Focus on the arrangement of objects/components and their relationships.

Use Case Diagrams

A Use Case Diagram in UML (Unified Modeling Language) is indeed used to visualize the relationships among actors (users or external entities) and use cases (the actions or services the system provides).

1. Actors:
    - Actors represent the roles played by users or external systems that interact with the system being modeled.
    - Actors are typically depicted as stick figures or blocks outside the system boundary.
    - They interact with the system by initiating and participating in one or more use cases.
2. Use Cases:
    - Use cases represent the functionality or behavior of the system from the perspective of its users (actors).
    - Each use case describes a sequence of actions or interactions between the user and the system to achieve a particular goal.
    - Use cases are depicted as ovals within the system boundary.
3. Relationships:
    - The primary relationship in a use case diagram is the association between actors and use cases.
    - An association between an actor and a use case indicates that the actor participates in the use case.
    - Each actor may be associated with one or more use cases, and vice versa.
    - Associations are typically represented by lines connecting actors to use cases.

Component Diagrams

A Component Diagram in the Unified Modeling Language (UML) is used to illustrate the organization and dependencies of components within a system. Here's a breakdown of the key aspects of Component Diagrams:

- **Components**: Represent modular parts of a system.
- **Interfaces**: Define contracts between components, showing provided and required services.
- **Dependencies**: Show relationships where one component relies on another.
- **Ports**: Points of interaction between a component and its environment.
- **Connectors**: Specify how components interact with each other or external systems.