Chapter 4:

Software Design, Architecture and Principles

4.1 Design Modeling Principles

Design modeling principles are essential guidelines that help software engineers create efficient, scalable, and maintainable systems. These principles, outlined by Roger S. Pressman, provide a structured approach to translating system requirements into a coherent design that guides the development process. By adhering to these principles, software designers ensure that the system not only meets its functional goals but also remains flexible, easy to understand, and adaptable to change. The principles focus on key aspects such as architecture, data, user interfaces, component independence, and iterative refinement, ensuring the development of high-quality software.

1. Design should be traceable to the requirements model

- The design must clearly link back to the system requirements, ensuring all specified needs are met without unnecessary features.

2. Always consider the architecture of the system to be built

- The overall architecture should be carefully considered from the beginning, providing a blueprint for organizing components and subsystems.

3. Design of data is as important as design of processing functions

- Both data structures and algorithms are critical. Proper data design ensures efficient data storage, retrieval, and manipulation.

4. Interfaces (both internal and external) must be designed with care

- Clear and well-defined interfaces between system components and external systems prevent integration problems and promote modularity.

5. User interface design should be tuned to the needs of the end user. However, in every case, it should stress ease of use

- The user interface (UI) should prioritize usability and be tailored to the user's needs, ensuring an intuitive and user-friendly experience.

6. Component-level design should be functionally independent

- Each software component should handle a specific function independently, making the system more modular and easier to maintain.

7. Components should be loosely coupled to one another and to the external environment

- Dependencies between components should be minimal, allowing each part to function without excessive reliance on others, improving flexibility and maintainability.

8. Design representations (models) should be easily understandable

- Design models should be simple, clear, and easy to comprehend for all stakeholders, ensuring effective communication and collaboration.

9. The design should be developed iteratively. With each iteration, the designer should strive for greater simplicity

- The design process should involve multiple iterations, refining and simplifying the design to improve clarity, performance, and maintainability with each cycle.

4.2 Design Process

The design process is a **crucial phase in software engineering that involves transforming requirements into a blueprint for constructing the software system.** According to Roger S. Pressman, a well-defined design process enhances the quality of the software product, facilitates communication among stakeholders, and provides a structured approach to problem-solving.

Objectives of the Design Process

- To create a blueprint for building software.
- To ensure that the system will meet its requirements and objectives.
- To facilitate communication and understanding among stakeholders.
- To provide a basis for implementation and testing.

Phases of the Design Process

1. **Conceptual Design**
   - Develop a high-level view of the system.
   - Identify major components and their interactions.
   - Create an architecture that reflects the overall structure of the system.
2. **Architectural Design**
   - Define the system's architecture, including:
     - Component organization.
     - Data flow.
     - Control flow.
   - Consider design patterns and architectural styles (e.g., layered architecture, client-server, microservices).
3. **Detailed Design**
   - Elaborate on the architectural design by specifying:
     - Data structures.
     - Algorithms.
     - Interfaces (both internal and external).
   - Focus on the specifics of how each component will be implemented.
4. **User Interface Design**
   - Design the user interface to ensure usability and meet user requirements.
   - Consider principles of human-computer interaction (HCI) to enhance user experience.
5. **Component-Level Design**
   - Define the functionality of each component.
   - Ensure that components are functionally independent and loosely coupled.
   - Consider error handling and exception management.
6. **Deployment Level Design**

   - Plan how the software will be deployed in the operational environment.
   - Consider aspects like server configurations, network requirements, installation procedures, and user access.
   - Ensure that deployment plans accommodate future scaling and updates.

TASK SET

Generic Task Set for Design

1. Examine the information domain model, and design appropriate data structures for data objects and their attributes.
2. Using the analysis model, select an architectural style that is appropriate for the software.
3. Partition the analysis model into design subsystems and allocate these subsystems within the architecture:
   - Be certain that each subsystem is functionally cohesive.
   - Design subsystem interfaces.
   - Allocate analysis classes or functions to each subsystem.
4. Create a set of design classes or components:
   - Translate analysis class description into a design class.
   - Check each design class against design criteria; consider inheritance issues.
   - Define methods and messages associated with each design class.
   - Evaluate and select design patterns for a design class or a subsystem.
   - Review design classes and revise as required.
5. Design any interface required with external systems or devices.
6. Design the user interface:
   - Review results of task analysis.
   - Specify action sequence based on user scenarios.
   - Create behavioral model of the interface.
   - Define interface objects, control mechanisms.
   - Review the interface design and revise as required.
7. Conduct component-level design.
   - Specify all algorithms at a relatively low level of abstraction.
   - Refine the interface of each component.
   - Define component-level data structures.
   - Review each component and correct all errors uncovered.
8. Develop a deployment model.

4.3 Architectural Design

Architectural design **is the process of defining a structured solution that meets all of the technical and operational requirements of a software system**. It **serves as a blueprint** for both the system and its components, guiding the implementation and ensuring that the final product is robust and efficient.

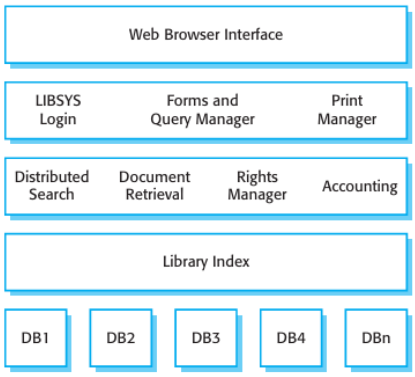

Figure: The architecture of the LIBSYS system.

Figure above is an example of how this layered architecture pattern can be applied to a library system called LIBSYS, which allows controlled electronic access to copyright material from a group of university libraries. This has a five-layer architecture, with the bottom layer being the individual databases in each library.

There are **various types of architecture design** such as:

a) Layered Architecture
b) Repository Architecture
c) Client-Server Architecture
d) The Pipe and Filter Pattern Architecture
e) MVC Architecture

a) Layered Architecture

The layered architecture pattern is another way of achieving separation and independence. Here, the system functionality is organized into separate layers, and each layer only relies on the facilities and services offered by the layer immediately beneath it. This layered approach supports the incremental development of systems. As a layer is developed, some of the services provided by that layer may be made available to users. The architecture is also changeable and portable. So long as its interface is unchanged, a layer can be replaced by another, equivalent layer. Furthermore, when layer interfaces change or new facilities are added to a layer, only the adjacent layer is affected. As layered systems localize machine

dependencies in inner layers, this makes it easier to provide multi-platform implementations of an application system. Only the inner, machine-dependent layers need be re-implemented to take account of the facilities of a different operating system or database.

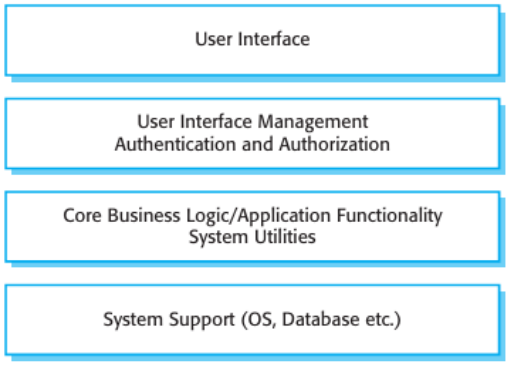

Figure: A generic layered architecture

Figure below is an example of a layered architecture with four layers. The lowest layer includes system support software—typically database and operating system support. The next layer is the application layer that includes the components concerned with the application functionality and utility components that are used by other application components. The third layer is concerned with user interface management and providing user authentication and authorization, with the top layer providing user interface facilities.

Characteristics of Layered Architecture

- **Separation of Concerns**: Each layer has distinct responsibilities, reducing complexity and allowing for easier maintenance.
- **Inter-layer Communication**: Layers communicate with adjacent layers. For instance, the presentation layer interacts with the application layer, which in turn communicates with the domain layer.
- **Encapsulation**: Each layer hides its implementation details from other layers, promoting modularity.
- **Reusability**: Components within each layer can be reused across different applications, increasing development efficiency.

Structure of Layered Architecture

1. **Presentation Layer (User Interface Layer)**:
    - This topmost layer is responsible for interacting with the end user.
    - It handles the presentation of data, user input, and user experience.
    - Functions include displaying information, taking user inputs, and invoking commands.
2. **Application Layer (Service Layer)**:
    - This layer contains the application logic that processes user requests.
    - It coordinates between the presentation layer and the lower layers.
    - Functions include processing commands, applying business rules, and managing data transactions.

3. **Domain Layer (Business Logic Layer)**:
    - This layer encapsulates the core business logic and rules of the application.
    - It is responsible for data processing, calculations, and decision-making.
    - Functions include validating data, enforcing business rules, and handling domain-specific logic.
4. **Data Layer (Persistence Layer)**:
    - This bottom layer manages data storage and retrieval.
    - It interacts with databases, file systems, or other data sources.
    - Functions include CRUD (Create, Read, Update, Delete) operations and data access methods.

Advantages of Layered Architecture

- **Maintainability**: Changes in one layer often do not affect other layers, simplifying updates and bug fixes.
- **Testability**: Layers can be tested independently, allowing for focused testing strategies.
- **Flexibility**: New features or layers can be added with minimal disruption to existing layers.
- **Scalability**: The architecture can be scaled horizontally or vertically by adding more layers or increasing the capacity of existing layers.

Disadvantages of Layered Architecture

- **Performance Overhead**: The multiple layers can introduce latency due to the overhead of communication between layers.
- **Rigidity**: In some cases, strict adherence to layer boundaries can limit design flexibility, making it harder to implement certain features.
- **Complexity**: While layers help manage complexity, they can also introduce additional complexity in terms of managing inter-layer interactions and dependencies.

Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.
Example	A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

b) Repository Architecture

Repository Architecture is a software architectural pattern that centralizes data storage, allowing multiple system components to interact with a shared data repository. It decouples data from application logic, facilitating data management and access. Repository Architecture is effective for managing data in complex systems, promoting flexibility and scalability while requiring careful consideration of performance and complexity.

The Repository pattern describes how a set of interacting components can share data. The majority of systems that use large amounts of data are organized around a shared database or repository. This model is therefore suited to applications in which data is generated by one component and used by another. Examples of this type of system include command and control systems, management information systems, CAD systems, and interactive development environments for software.

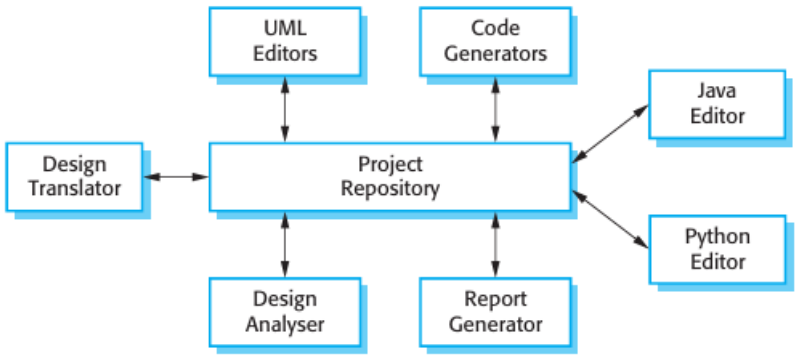

Figure: A repository architecture for an IDE

Figure above is an illustration of a situation in which a repository might be used. This diagram shows an IDE that includes different tools to support model-driven development.

Key Components

1. **Repository**: Central data store (e.g., database) that holds all application data.
2. **Components/Modules**: Various system components that interact with the repository to retrieve and manipulate data.
3. **Data Access Layer**: An intermediary layer that abstracts data interactions and provides methods for querying and modifying data.

Characteristics

- **Centralized Data Management**: Ensures consistency and integrity of data.
- **Loose Coupling**: Components are independent of each other regarding data access.
- **Abstraction**: Simplifies data interactions through a dedicated data access layer.

- **Shared Data**: Allows multiple components to access and modify the same data.

Advantages

- **Flexibility**: Easy to change data structures without affecting components.
- **Scalability**: Can scale with additional components and optimized repositories.
- **Reusability**: The data access layer can be reused across applications.
- **Data Consistency**: Central management helps maintain data integrity.

Disadvantages

- **Performance Bottlenecks**: Central repository may slow down with high access frequency.
- **Complexity**: Can introduce complexity in managing component interactions.
- **Single Point of Failure**: Repository issues can affect the entire system.

Use Cases

- **Enterprise Applications**: ERP and CRM systems that require centralized data management.
- **Data-Driven Applications**: Analytics platforms that handle large data volumes.
- **Microservices**: A shared data store for microservices needing access to common data.

c) Client Server Architecture

Client-Server Architecture is a distributed computing model where tasks are divided between service providers (servers) and service requesters (clients). This architecture enables communication and resource sharing over a network.

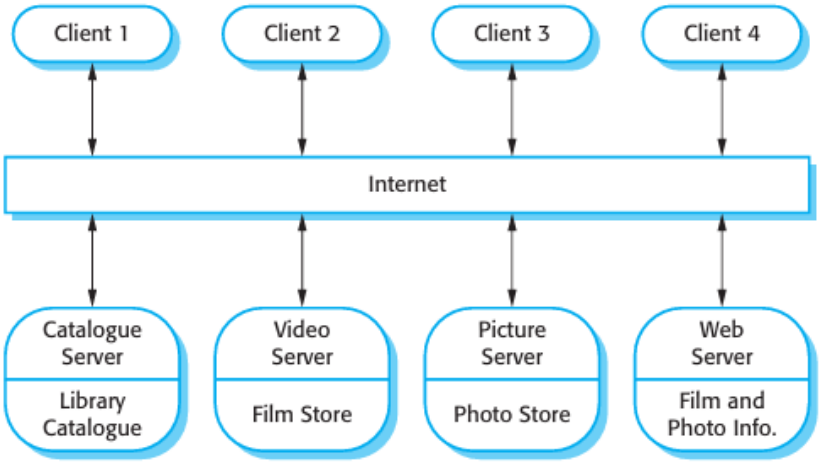

Figure 6.11 A client–server architecture for a film library

Figure above is an example of a system that is based on the client–server model. This is a multi-user, web-based system for providing a film and photograph library. In this system, several servers manage and display the different types of media. Video frames need to be transmitted quickly and in synchrony but at relatively low resolution. They may be compressed in a store, so the video server can handle video compression and decompression in different formats. Still pictures, however, must be maintained at a high resolution, so it is appropriate to maintain them on a separate server.

Key Components

1. **Client**:
   - User-facing application that requests services from the server (e.g., web browsers, mobile apps).
2. **Server**:
   - A powerful system that provides services and processes requests from clients (e.g., web servers, database servers).

Characteristics

- **Separation of Concerns**: Distinct roles for clients and servers.
- **Scalability**: Servers can be scaled up or out to handle more clients.
- **Network Communication**: Clients and servers communicate over networks using protocols (e.g., HTTP, TCP/IP).

Advantages

- **Distributed Architecture:** effective use can be made of networked systems with many distribted processors.
- **Resource Sharing**: Multiple clients can share a server's resources.
- **Centralized Management**: Easier updates and maintenance for servers.
- **Flexibility**: Clients can be independently developed across different platforms.

Disadvantages

- **Single Point of Failure**: Server downtime affects all clients.
- **Network Dependency**: Performance can be impacted by network issues.
- **Security Risks**: Exposing servers to networks can introduce vulnerabilities.

Use Cases

- **Web Applications**: Browsers fetching web pages from servers.
- **Database Access**: Client applications retrieving data from databases.
- **File Sharing**: Accessing files from shared file servers.

Name	Client–server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

d) Pipe and Filter Architecture

Pipe and Filter Architecture is a software design pattern used to process streams of data. It breaks down a data processing task into a series of discrete processing steps (filters), which are connected by channels (pipes). Each filter processes data and passes the output to the next filter through the pipes.

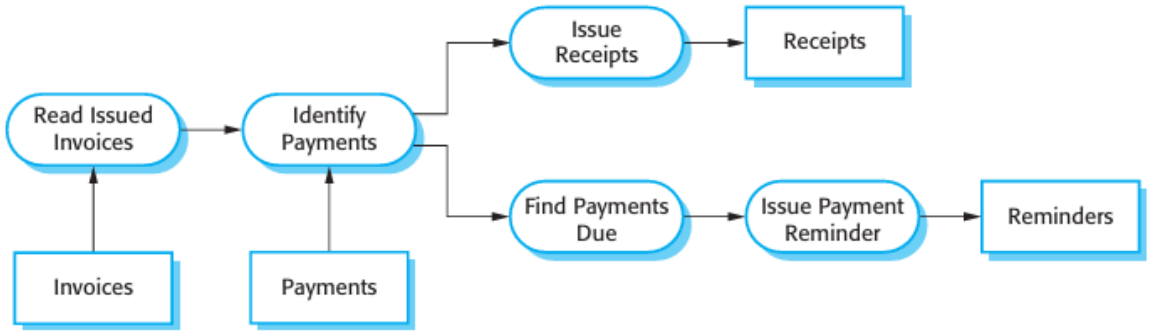

Figure : an example of the Pipe and Filter architecture

An example of this type of system architecture, used in a batch processing appli cation, is shown in Figure 6.13. An organization has issued invoices to customers. Once a week, payments that have been made are reconciled with the invoices. For those invoices that have been paid, a receipt is issued. For those invoices that have not been paid within the allowed payment time, a reminder is issued.

Key Components

1. **Filters**:
   - Independent processing units that transform input data into output data.
   - Each filter operates on the data it receives, performs its processing, and sends the results to the next filter.
   - Filters can be reused and combined in various ways.
2. **Pipes**:
   - Channels that connect filters and transport data between them.
   - They define the flow of data from one filter to another.
   - Pipes can handle different data formats and can provide buffering to manage data flow.

Characteristics

- **Decoupled Components**: Filters are independent; changes to one filter do not affect others.
- **Data Stream Processing**: Data flows continuously through the system, making it suitable for handling streams of information.
- **Reusability**: Filters can be reused in different configurations or applications.

Advantages

- **Modularity**: Easy to add, remove, or replace filters without affecting the overall system.
- **Flexibility**: New processing steps can be introduced easily, allowing for dynamic reconfiguration.
- **Parallel Processing**: Filters can operate concurrently, improving performance and efficiency.

Disadvantages

- **Latency**: Data may experience delays as it passes through multiple filters.
- **Complexity**: Managing the connections and data flow between numerous filters can be complex.
- **Error Handling**: Implementing error handling can be challenging, as it needs to be coordinated across filters.

Use Cases

- **Data Processing Pipelines**: Used in data transformation tasks, such as ETL (Extract, Transform, Load) processes.
- **Streaming Applications**: Suitable for applications that process continuous data streams, like video processing or audio processing.
- **Compilers**: The stages of a compiler (lexical analysis, parsing, code generation) can be represented as filters.

Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.13 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

4.4 Interface, Component, Database Design

Interface Design:

Interface design is a critical aspect of software engineering that focuses on the interaction between users and software systems. Interface design refers to the process of defining how users interact with a software system, including the layout, controls, and overall user experience. A well-designed interface improves usability, increases user satisfaction, and minimizes the potential for errors during interactions.

Interface Design Process

1. **User Analysis**:
   - Understand the target audience, including their needs, preferences, and technical abilities. Conduct user research, surveys, and usability testing.
2. **Task Analysis**:
   - Analyze the tasks users will perform using the system. Identify the steps required to complete each task and design the interface to facilitate these actions.
3. **Prototyping**:
   - Create low-fidelity and high-fidelity prototypes of the interface. This allows for iterative testing and feedback from users before finalizing the design.
4. **Usability Testing**:
   - Conduct usability testing with real users to identify any issues or areas for improvement. Gather feedback and make necessary adjustments to enhance the interface.
5. **Implementation and Evaluation**:
   - Implement the final design in the software. After deployment, continue to evaluate the interface based on user feedback and usage data, making updates as needed.

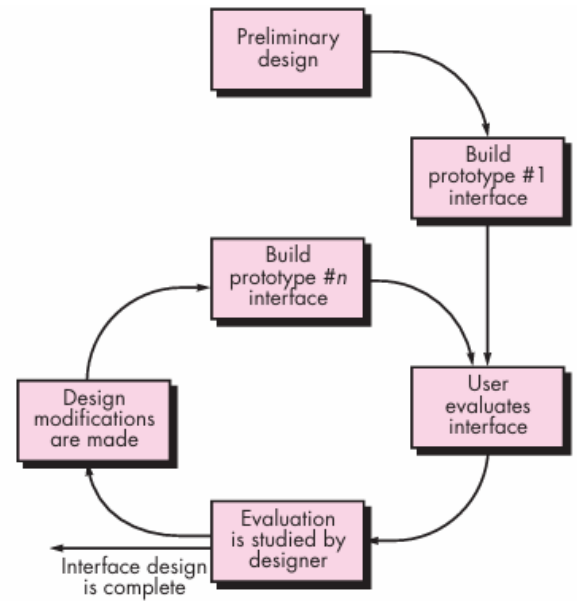

Figure: The Interface Evaluation Cycle

Principles of Effective Interface Design

Sommerville emphasizes several principles:

1. **Consistency:**
   - Maintain uniformity in interface design to avoid confusion.
   - For UIs, this includes consistent layouts and interaction styles.
2. **Clarity:**
   - Ensure the interface is self-explanatory.
   - Use descriptive names for APIs, meaningful icons in GUIs, and clear error messages.
3. **Flexibility:**
   - Support different levels of user expertise (e.g., novice vs. advanced users).
   - Allow for different data input methods or interaction patterns.
4. **Fault Tolerance:**
   - Design interfaces to handle unexpected scenarios without crashing.
   - Provide clear error messages or recovery options.
5. **Minimize Dependencies:**
   - Avoid tightly coupling components to maintain modularity.
   - Use abstraction layers to hide implementation details.

Component Design:

Component design focuses on creating reusable and maintainable software components that can be integrated into larger systems. Component design involves defining the internal structure and behavior of software components that can be independently developed, tested, and reused within various applications.

Component Design Process

1. **Identify Components**:
   - Analyze the system requirements to identify potential components. Group related functionalities into cohesive components.
2. **Define Interfaces**:
   - Clearly define the interfaces for each component, specifying the input and output requirements, as well as any dependencies on other components.
3. **Design Internal Structure**:
   - Determine the internal structure and behavior of each component. This may include algorithms, data structures, and any necessary state management.
4. **Implement Components**:
   - Write the code for each component, ensuring adherence to design principles and standards. Use appropriate programming languages and tools.
5. **Test Components**:
   - Conduct unit testing for each component to verify functionality and identify any defects. Ensure that components behave as expected when integrated with others.
6. **Documentation**:

- Create documentation for each component, including its purpose, interfaces, usage examples, and any design considerations. This is essential for future maintenance and reuse.

7. **Integration**:
   - Integrate components into the overall system, ensuring that they work together as intended. Conduct integration testing to verify interactions between components.

Database Design:

Database design is a critical aspect of software engineering that focuses on defining the structure, relationships, and constraints of data within a database. It emphasizes the importance of effective database design in ensuring data integrity, efficiency, and accessibility within software applications. Database design involves creating a detailed plan for the structure of a database, including the organization of data, relationships among data elements, and constraints to maintain data integrity. A well-designed database enhances data consistency, supports efficient data retrieval, and reduces redundancy, leading to improved application performance.

Database Design Process

1. **Requirements Gathering**:
   - Collect requirements from stakeholders to understand the data needs of the application. This includes identifying key entities and their relationships.
2. **Conceptual Design**:
   - Create a conceptual model using tools like Entity-Relationship diagrams. Define entities, attributes, and relationships without worrying about implementation details.
3. **Logical Design**:
   - Transform the conceptual model into a logical model that defines the structure of the database. Specify tables, columns, data types, and constraints.
4. **Physical Design**:
   - Design the physical structure of the database, considering how data will be stored and accessed on storage devices. This includes indexing strategies and storage allocation.
5. **Implementation**:
   - Implement the database using a Database Management System (DBMS). Create tables, define relationships, and establish constraints as per the design.
6. **Testing**:
   - Test the database to ensure it meets the requirements and functions as expected. Perform validation checks and run test queries to assess performance.
7. **Documentation**:
   - Document the database design, including schemas, data models, and any important design decisions. This aids in future maintenance and updates.
8. **Maintenance**:
   - Monitor the database performance and make necessary adjustments over time. Regularly update documentation and perform backups.

4.5 Design Patterns

Design Patterns are general reusable solutions to common problems in software design. They represent best practices that can be applied in various situations, promoting code reusability, maintainability, and scalability.

Categories of Design Patterns

1. **Creational Patterns**:
   - Deal with object creation mechanisms, providing increased flexibility and reuse of existing code.
   - **Examples**:
     - **Singleton**: Ensures a class has only one instance and provides a global point of access.
     - **Factory Method**: Defines an interface for creating objects but allows subclasses to alter the type of objects that will be created.
     - **Abstract Factory**: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
2. **Structural Patterns**:
   - Concerned with how objects are composed to form larger structures, focusing on the relationships between entities.
   - **Examples**:
     - **Adapter**: Allows incompatible interfaces to work together by converting the interface of a class into another interface.
     - **Decorator**: Adds behavior or responsibilities to individual objects dynamically without affecting the behavior of other objects.
     - **Facade**: Provides a simplified interface to a complex subsystem, making it easier to use.
3. **Behavioral Patterns**:
   - Focus on communication between objects, defining how objects interact and how responsibilities are assigned.
   - **Examples**:
     - **Observer**: Defines a one-to-many dependency between objects, so when one object changes state, all its dependents are notified.
     - **Strategy**: Defines a family of algorithms, encapsulating each one and making them interchangeable.
     - **Command**: Encapsulates a request as an object, allowing for parameterization and queuing of requests.

Benefits of Using Design Patterns

- **Reusability**: Patterns can be reused across different projects and applications.
- **Maintainability**: Well-defined patterns make it easier to understand and modify code.
- **Communication**: Patterns provide a common vocabulary for developers, facilitating clearer communication about design decisions.

4.6 Security By Design

Security by Design is a critical concept in software engineering that emphasizes integrating security measures throughout the software development lifecycle. This approach ensures that systems are resilient to threats and vulnerabilities from the outset, rather than treating security as an afterthought.

"Secure-by-design" is an approach to designing and building systems with security in mind. **It involves implementing continuous security practices, tools, and controls from the beginning of the software development lifecycle, ensuring that products are inherently secure.** Embedding security measures early in the development lifecycle drastically reduces the attack surface for potential exploits – meaning products hit the market with a strong security posture.

Principles for Security by Design

1. **Least Privilege**:
   - Ensure that users and systems have only the permissions necessary to perform their functions. This limits the potential impact of a security breach.
2. **Defense in Depth**:
   - Implement multiple layers of security controls and safeguards. If one layer fails, others will still provide protection, reducing the risk of a complete system compromise.
3. **Secure Defaults**:
   - Configure systems with secure settings out-of-the-box. Default configurations should prioritize security to minimize vulnerabilities from user negligence.
4. **Fail-Safe Defaults**:
   - In the event of a failure, systems should revert to a secure state. This prevents the exposure of sensitive data or functionalities during unforeseen incidents.
5. **Input Validation**:
   - Rigorously validate all incoming data to ensure it meets expected formats and criteria. This protects against common vulnerabilities like injection attacks and buffer overflows.
6. **Security Audits and Testing**:
   - Conduct regular security assessments, including code reviews and penetration testing, to identify and address vulnerabilities early in the development process.
7. **Modular Design**:
   - Design systems with modular components that can be individually secured and updated. This approach allows for greater flexibility in enhancing security without overhauling the entire system.
8. **Cryptography**:
   - Use strong encryption techniques to protect sensitive data in transit and at rest. This ensures confidentiality and integrity against unauthorized access.
9. **Logging and Monitoring**:
   - Implement comprehensive logging of security events and system activities. This facilitates monitoring for suspicious behavior and aids in incident response.
10. **User Awareness and Training**:

- Educate users and developers on security best practices. Awareness reduces the likelihood of human error that could lead to security breaches.

4.7 Object Oriented Design

Object-Oriented Design (OOD) refers to the process of designing a system or application using object-oriented principles. It involves identifying the system's objects, their interactions, and how they collaborate to solve a problem. OOD is a key phase in object-oriented software development and serves as a bridge between requirements analysis and implementation.

Key Concepts of Object-Oriented Analysis

1. **Objects**
   - Objects are entities that combine data (attributes) and behavior (methods).
   - Represent real-world entities (e.g., a "Student" in a university system) or abstractions (e.g., "Transaction" in a banking system).
2. **Classes**
   - Classes are templates for creating objects.
   - They define common attributes and methods for objects of the same type.
3. **Encapsulation**
   - Hides the internal details of an object and exposes only what is necessary through public interfaces.
4. **Inheritance**
   - Allows objects to acquire properties and behaviors from other objects.
   - Promotes code reuse and hierarchical organization.
5. **Polymorphism**
   - Ability of different classes to respond to the same message (method call) in different ways.
   - Facilitates dynamic behavior by overriding or overloading methods.

Steps in Object-Oriented Analysis

1. **Understand the Problem Domain**
   - Identify the real-world entities and their interactions.
   - Focus on understanding the functional requirements of the system.
2. **Identify Objects and Classes**
   - Extract potential objects and classes from the problem domain by analyzing nouns in requirements specifications or use cases.
   - Example: In a library management system, objects might include "Book," "Member," and "Loan."
3. **Define Attributes and Methods**
   - For each object, define its attributes (data it holds) and methods (operations it performs).
   - Example: For a "Book" class:
     - Attributes: Title, Author, ISBN.
     - Methods: Borrow(), Return().
4. **Establish Relationships**
   - Identify how objects interact and relate to one another.

- Common relationships include:
  - **Association**: Represents a general connection between objects.
  - **Aggregation**: Represents a "whole-part" relationship (e.g., "Department" contains "Employees").
  - **Inheritance**: Represents an "is-a" relationship (e.g., "Car" is a "Vehicle").

5. **Define Scenarios and Use Cases**
   - Use cases describe the interactions between users (actors) and the system.
   - Scenarios outline specific examples of these interactions.
6. **Model the System**
   - Create visual representations of the system using diagrams:
     - **Class Diagrams**: Define objects, classes, and their relationships.
     - **Sequence Diagrams**: Show how objects interact over time.
     - **State Diagrams**: Represent the states and transitions of objects.

Deliverables of Object-Oriented Design

1. **Object Model:**
   - A structured model showing objects, their attributes, methods, and relationships.
2. **Use Case Diagrams:**
   - Illustrate how users interact with the system.
3. **Class Diagrams:**
   - Show the classes in the system and their relationships.
4. **Sequence Diagrams:**
   - Demonstrate the sequence of interactions between objects to accomplish a specific task.
5. **State Diagrams:**
   - Represent the state transitions of critical objects in response to events.

Assignment:

What is Object Oriented Analysis? Explain

4.8 Embedded System Design

Embedded systems are computer systems that are part of larger systems and are designed to perform dedicated functions. They include hardware and software components.

Embedded system design involves creating specialized computing systems that are dedicated to performing specific tasks within larger mechanical or electrical systems. These systems are characterized by their constraints on resources (such as power, memory, and processing speed) and their integration into other devices.

Characteristics of Embedded Systems:

- **Real-Time Operation**: Must respond to events within strict timing constraints.
- **Resource Constraints**: Limited processing power, memory, and energy consumption.
- **Reliability and Stability**: Must operate continuously and reliably over long periods.
- **Interactivity**: Often interface with users or other systems, requiring effective communication protocols.

1. **Components of Embedded Systems**:
    - **Microcontrollers/Microprocessors**: The central processing unit (CPU) that executes instructions.
    - **Memory**: Includes both volatile (RAM) and non-volatile (Flash, EEPROM) storage for program code and data.
    - **Input/Output Interfaces**: Mechanisms for the system to interact with the external environment, including sensors and actuators.

- **Power Supply**: Provides energy to the embedded system, often requiring efficient power management techniques.

Design Considerations

1. **Performance**: Ensure the system meets real-time performance requirements.
2. **Power Consumption**: Optimize for low power usage, especially for battery-operated devices.
3. **Cost**: Design within budget constraints, balancing performance with cost-effectiveness.
4. **Scalability**: Design for potential future enhancements or changes in requirements.
5. **Security**: Incorporate security measures to protect against unauthorized access and vulnerabilities.

Design Process of Embedded Systems

1. **Requirement Analysis**:
   - Understand and specify the functionality, performance, and constraints of the system.
2. **System Architecture Design**:
   - Define the overall system architecture, including hardware and software components. Choose suitable microcontrollers and design interfaces.
3. **Hardware Design**:
   - Design the hardware components, including circuit design, PCB layout, and selection of peripherals.
4. **Software Development**:
   - Develop the software, which may include embedded operating systems, device drivers, and application code. Programming languages often used include C, C++, and assembly language.
5. **Integration and Testing**:
   - Integrate hardware and software components and perform thorough testing to ensure functionality, performance, and reliability.
6. **Debugging and Validation**:
   - Use debugging tools to identify and fix issues. Validate the system against requirements and ensure it meets all performance criteria.
7. **Deployment**:
   - Deploy the embedded system in its intended environment, ensuring proper installation and configuration.
8. **Maintenance and Updates**:
   - Provide ongoing support and updates as needed, including bug fixes and performance enhancements.