Basic Concepts in Software Design

Prof. K. P. Hewagamage

What is design?

- Design is the process of creating a plan or solution to meet specific requirements or solve a problem. It involves making decisions about the structure, components, and aesthetics of a product or system.

Prerequisites for Software Design

1. Requirement Analysis

- **Understanding System Requirements:**
  - Functional and non-functional requirements.
  - Stakeholder needs and expectations.

2. Domain Knowledge

- **Familiarity with the Industry:**
  - Context-specific standards and practices.
  - Insights into the specific domain.

3. Technical Proficiency

- **Knowledge of Software Engineering Principles:**
  - Programming languages and frameworks.
  - Design patterns and architecture styles.

Prerequisites for Software Design

4. Problem-Solving Skills

- **Creative and Critical Thinking:**
  - Identifying challenges and devising solutions.
  - Analytical evaluation of design options.

5. Communication Skills

- **Effective Interaction:**
  - Gathering requirements and sharing design ideas.
  - Ensuring alignment with stakeholders.

6. Project Constraints Awareness

- **Understanding Constraints:**
  - Budget, timeline, and resource limitations.
  - Technical feasibility considerations.

Challenges in Software Design

1. Balancing Competing Requirements

- **Functional vs. Non-functional Needs:**
  - Ensuring features work as intended while meeting performance, security, and usability standards.

2. Managing Complexity

- **Large-Scale Systems:**
  - Handling numerous components, modules, and interactions.
  - Maintaining a clear and understandable structure.

3. Dealing with Ambiguity

- **Unclear or Changing Requirements:**
  - Adapting to evolving user needs and business goals.
  - Managing uncertainty in project scope and objectives.

4. Ensuring Scalability and Flexibility

- **Future Growth and Changes:**
  - Designing systems that can handle increased loads and new features.
  - Avoiding rigid structures that hinder adaptation.

Challenges in Software Design

5. Meeting Deadlines and Budget Constraints

- **Time and Cost Management:**
  - Delivering a viable product within the given timeframe and budget.
  - Prioritizing features and functionalities effectively.

6. Integrating with Existing Systems

- **Compatibility and Interoperability:**
  - Ensuring new designs work seamlessly with legacy systems and third-party services.

7. Maintaining Quality

- **Ensuring Reliability and Performance:**
  - Preventing defects and ensuring the system meets quality standards.

Key Components in Software Design

1. Architectural Design

- **System Structure:**
  - Defines the high-level structure of the system.
  - Includes the overall system architecture, layers, and components.

2. Component Design

- **Modularization:**
  - Breaks down the system into smaller, manageable modules or components.
  - Specifies the internal structure and functionality of each component.

3. Interface Design

- **User Interface (UI):**
  - Focuses on the interaction between the user and the system.
  - Includes layout, navigation, and user experience considerations.
- **System Interfaces:**
  - Defines how different system components communicate and interact.
  - Includes APIs, data formats, and communication protocols.

4. Data Design

- **Data Structures and Databases:**
  - Specifies how data is stored, organized, and accessed.
  - Includes data models, schemas, and data flow.

Key Components in Software Design

5. Security Design

- **Security Mechanisms:**
  - Identifies potential security threats and vulnerabilities.
  - Defines authentication, authorization, encryption, and data protection measures.

6. Performance Design

- **Efficiency and Scalability:**
  - Ensures the system can handle expected loads and performance requirements.
  - Includes load balancing, caching, and optimization strategies.

7. Design Documentation

- **Blueprints and Specifications:**
  - Provides detailed descriptions and diagrams of the system design.
  - Includes UML diagrams, flowcharts, and design documents.

Software design differs significantly between Agile and Plan-Driven

1. Design Approach

- **<u>Agile Design:</u>**
- **Incremental and Iterative:**
  - Design evolves over time with each iteration. Small, incremental changes are made as new requirements are discovered.
- **Just-In-Time (JIT) Design:**
  - Design is done "just in time," often at the last responsible moment, to accommodate changes and avoid waste.
- **Collaborative and Adaptive:**
  - Design involves continuous collaboration among team members and stakeholders. It adapts to changing requirements and feedback.

- **<u>Plan-Driven Design:</u>**
- **Big Design Up Front (BDUF):**
  - Extensive design is done at the start of the project. The design is intended to be complete and fully specified before implementation begins.
- **Predictive and Structured:**
  - The design process follows a well-defined sequence of steps. It relies on comprehensive documentation and strict adherence to plans.
- **Limited Flexibility:**
  - Changes to the design are difficult and costly once development has begun, as they require formal change requests and approvals.

Software design differs significantly between Agile and Plan-Driven

- **2. Documentation and Specification**

- **Agile Design:**
- **Minimal Documentation:**
  - Emphasizes working software over comprehensive documentation. Documentation is typically light and focuses on essential aspects.
- **User Stories and Diagrams:**
  - User stories, use cases, and lightweight diagrams are used to convey design decisions and requirements.

- **Plan-Driven Design:**
- **Comprehensive Documentation:**
  - Extensive design documents, including detailed specifications, diagrams, and technical requirements, are created and maintained.
- **Formal Specifications:**
  - Design specifications are detailed and formalized to ensure clarity and completeness.

Software design differs significantly between Agile and Plan-Driven

- **3. Design Flexibility and Change Management**

Agile Design:

- **High Flexibility:**
  - Agile design is inherently flexible, allowing for changes in requirements and design throughout the development process.
- **Continuous Feedback and Improvement:**
  - Design is continuously refined based on feedback from iterations, stakeholder reviews, and testing.

Plan-Driven Design:

- **Low Flexibility:**
  - Changes to the design are challenging due to the upfront commitment to a comprehensive plan. Changes often require formal change management processes.
- **Fixed Scope:**
  - The design is based on a fixed set of requirements established during the planning phase, with limited scope for changes.

Software design differs significantly between Agile and Plan-Driven

- **4. Team Involvement and Collaboration**

Agile Design:

- **Cross-Functional Teams:**
  - Agile teams are typically cross-functional, including developers, designers, testers, and other stakeholders, all working closely together.
- **Frequent Collaboration:**
  - Regular meetings, such as daily stand-ups and sprint reviews, facilitate continuous communication and collaboration.

Plan-Driven Design:

- **Specialized Roles:**
  - Teams are often organized into specialized roles, with distinct phases for requirements, design, development, and testing.
- **Formal Communication:**
  - Communication often follows formal channels, with defined hand-offs between phases and roles.

Software design differs significantly between Agile and Plan-Driven

- **5. Risk Management**

Agile Design:

- **Early and Continuous Risk Mitigation:**
  - Risks are addressed early and continuously throughout the project, as the iterative nature allows for regular reassessment and adjustment.

Plan-Driven Design:

- **Risk Assessment at the Beginning:**
  - Risks are identified and planned for during the initial design phase. Mitigation strategies are incorporated into the plan but may be harder to adjust later.

Major Approaches in Software Design

1. Procedural Design

- **Focus:**
  - Based on procedures or routines.
  - Emphasizes a sequence of computational steps.
- **Characteristics:**
  - Functions and subroutines.
  - Linear flow of control.

2. Object-Oriented Design (OOD)

- **Focus:**
  - Based on objects representing data and behavior.
  - Uses principles like encapsulation, inheritance, and polymorphism.
- **Characteristics:**
  - Classes and objects.
  - Focus on reusable and modular components.

Major Approaches in Software Design

3. Functional Design

- **Focus:**
  - Based on mathematical functions.
  - Emphasizes immutability and statelessness.
- **Characteristics:**
  - Pure functions without side effects.
  - Higher-order functions and function composition.

4. Component-Based Design

- **Focus:**
  - Based on building systems from reusable components.
  - Emphasizes separation of concerns and modularity.
- **Characteristics:**
  - Well-defined interfaces.
  - Independent, interchangeable components.

Major Approaches in Software Design

5. Event-Driven Design

- **Focus:**
  - Based on responding to events or changes in state.
  - Often used in UI design and real-time systems.
- **Characteristics:**
  - Event handlers and listeners.
  - Asynchronous communication.

6. Service-Oriented Design (SOA)

- **Focus:**
  - Based on designing software as a collection of services.
  - Emphasizes loose coupling and interoperability.
- **Characteristics:**
  - Web services and APIs.
  - Service contracts and messaging.

Key Aspects of Modeling in Software Design:

Abstraction:
•Simplifying complex systems by focusing on essential aspects while omitting unnecessary details.

Visualization:
•Providing visual representations such as diagrams and flowcharts to illustrate system components and their interactions.

Communication:
•Serving as a common language among stakeholders, including developers, designers, clients, and users.

Analysis and Specification:
•Enabling analysis of system requirements, design decisions, and potential risks.

Guidance for Implementation:
•Acting as a guide for the development team during the coding phase, ensuring consistency and alignment with the design.

Common Modeling Techniques:

Unified Modeling Language (UML):
•A standardized visual language for creating diagrams such as class diagrams, use case diagrams, sequence diagrams, and more.

Entity-Relationship Diagrams (ERD):
•Used to model data relationships in databases.

Flowcharts:
•Visual representations of workflows or processes.

Data Flow Diagrams (DFD):
•Illustrate the flow of data within a system.

State Diagrams:
•Show the states of a system and transitions between them based on events.

Modeling is a crucial part of the software design process, providing clarity, improving communication, and reducing the risk of errors during development

Importance of Modeling at the Beginning of Software Design

Clarification of Requirements:
•Modeling helps translate complex requirements into clear, visual representations, making it easier for stakeholders to understand and validate the system's requirements.

Visualization of the System:
•Early models provide a visual overview of the system's structure and behavior, helping to identify potential issues and inconsistencies before implementation begins.

Communication Tool:
•Models serve as a common language among project team members and stakeholders, facilitating clear and effective communication about the system's design and functionality.

Decision Making:
•By visualizing different aspects of the system, modeling supports informed decision-making about architecture, components, data flows, and interfaces.

Importance of Modeling at the Beginning of Software Design

Risk Mitigation:
•Early detection of design flaws, ambiguities, or missing requirements can be addressed before they become costly problems during later stages of development.

Guidance for Development:
•Models provide a roadmap for developers, offering detailed guidance on how to implement the system's features and ensuring alignment with the overall design.

Documentation:
•Modeling contributes to project documentation, creating a reference for future maintenance and enhancements

Modeling for Software Architecture

1. Visual Representation of System Structure

- **Architectural Diagrams:**
  - Models such as component diagrams, deployment diagrams, and layered architecture diagrams visually represent the system's structure. These diagrams help in identifying the key components, their interactions, and how they fit together within the overall system.

2. Clarifying System Requirements and Constraints

- **Use Case and Requirements Modeling:**
  - Use case diagrams and other requirement models help clarify what the system must do. By understanding the functional and non-functional requirements, architects can make informed decisions about the necessary architectural components and their relationships.

3. Identifying Key Components and Interfaces

- **Component and Class Diagrams:**
  - Modeling helps identify the key components (modules, services, etc.) and their responsibilities. Class diagrams can also specify the attributes and operations of these components, along with their interfaces, ensuring that interactions are well-defined.

4. Analyzing System Interactions and Communication

- **Sequence and Communication Diagrams:**
  - These diagrams model the flow of information and the interaction between components. They help in understanding how data moves through the system, the order of operations, and the communication protocols required.

Modeling for Software Architecture

5. Evaluating Design Alternatives

- **Architectural Patterns and Styles:**
  - Modeling allows for the exploration of different architectural patterns (e.g., client-server, microservices, layered) and styles (e.g., RESTful, event-driven). By visualizing these options, architects can evaluate the pros and cons of each approach and choose the most suitable one.

6. Ensuring Scalability and Performance

- **Performance and Load Modeling:**
  - Models can simulate the system's behavior under different conditions, helping to predict its performance and scalability. This analysis can influence architectural decisions, such as the need for load balancing, caching, or distributed systems.

7. Facilitating Stakeholder Communication and Consensus

- **Unified Modeling Language (UML) and Other Notations:**
  - Standardized modeling languages and notations ensure that all stakeholders, including developers, business analysts, and customers, have a shared understanding of the architecture. This common understanding is crucial for gaining consensus on architectural decisions.

8. Supporting Risk Analysis and Mitigation

- **Risk Identification:**
  - By visualizing the system's structure and its dependencies, modeling helps identify potential risks, such as single points of failure or security vulnerabilities. Architects can then design mitigation strategies, such as redundancy or security layers, into the architecture.

Modeling and detail design

1. Specifying Component Interfaces

- **Interface Definition:**
  - Models such as class diagrams and interface diagrams clearly define the interfaces of components, specifying the methods, parameters, return types, and access controls. This ensures that developers understand how components interact and can implement consistent interfaces.

2. Detailing Internal Component Structure

- **Class and Object Diagrams:**
  - These diagrams describe the internal structure of each component, including attributes, methods, relationships, and dependencies. They provide a blueprint for implementing the internal logic and data structures.

3. Clarifying Component Interactions

- **Sequence and Communication Diagrams:**
  - These models illustrate how objects and components interact over time, showing the flow of messages and data. This helps in understanding the dynamic behavior of the system and ensures that the interactions align with the system's requirements.

4. Defining Data Models

- **Entity-Relationship Diagrams (ERD) and Data Flow Diagrams (DFD):**
  - ERDs and DFDs define the data structures, relationships, and data flow within the system. They help in designing databases, data access layers, and ensuring data integrity and consistency.

5. Ensuring Consistency and Integrity

- **Consistency Checks:**
  - Modeling helps in maintaining consistency across different parts of the system. For example, ensuring that the data model aligns with the application's business logic and user interface.

Modeling and detail design

6. Facilitating Code Generation and Implementation

- **Code Generation:**
  - In some cases, detailed models can be used to generate boilerplate code, reducing manual coding effort and minimizing errors. This is particularly useful for repetitive tasks, such as data access or service layer implementation.

7. Providing Documentation and Reference

- **Design Documentation:**
  - Detailed models serve as comprehensive documentation for developers, testers, and maintainers. They provide a clear reference for understanding the system's design and its components, aiding in onboarding new team members and future maintenance.

8. Supporting Design Patterns and Best Practices

- **Incorporating Design Patterns:**
  - Modeling can illustrate the application of design patterns, such as Singleton, Factory, or Observer, within the system. This ensures that best practices are followed, promoting code reuse, flexibility, and maintainability.

9. Validating and Verifying Design Choices

- **Verification and Validation:**
  - Detailed models enable thorough review and validation of the design, helping to identify potential issues or discrepancies early. This reduces the risk of costly changes later in the development process.

10. Enhancing Communication Among Teams

- **Cross-Functional Understanding:**
  - Models provide a visual and standardized way to communicate complex design details among different teams, including developers, testers, project managers, and stakeholders. This fosters a shared understanding and alignment on the system's design.

The flow of software design, starting from the Software Requirements Specification (SRS) document,

1. Requirement Analysis and Validation

- **Review SRS Document:**
  - Understand the functional and non-functional requirements.
- **Identify Constraints:**
  - Acknowledge technical, business, and regulatory constraints.
- **Clarify Ambiguities:**
  - Address any unclear or incomplete requirements with stakeholders.

2. High-Level Design (HLD) / Architectural Design

- **Define System Architecture:**
  - Decide on the architectural style (e.g., layered, microservices, client-server).
- **Identify Major Components:**
  - Outline the main modules, subsystems, and their responsibilities.
- **Define Component Interactions:**
  - Specify how components communicate (e.g., APIs, data flows).
- **Create Architectural Diagrams:**
  - Use models like component diagrams, deployment diagrams, and data flow diagrams.

The flow of software design, starting from the Software Requirements Specification (SRS) document,

3. Detailed Design (Low-Level Design, LLD)

- **Component-Level Design:**
  - Detail the internal structure of each component, including classes, methods, and data structures.
- **Interface Design:**
  - Define the interfaces and protocols for component interaction.
- **Data Design:**
  - Specify the database schema, data formats, and data flow.
- **User Interface Design:**
  - Create wireframes and mockups for the user interface, if applicable.
- **Behavioral Modeling:**
  - Develop sequence diagrams, state diagrams, and activity diagrams to model dynamic behaviors.

4. Design Documentation

- **Compile Design Documents:**
  - Prepare comprehensive documentation, including design rationale, assumptions, and decisions.
- **Document Design Patterns:**
  - Specify any design patterns used and their application in the system.

The flow of software design, starting from the Software Requirements Specification (SRS) document,

5. Design Review and Validation

- **Internal Review:**
  - Conduct peer reviews and walk-throughs of the design documents.
- **Stakeholder Validation:**
  - Validate the design with stakeholders, ensuring alignment with requirements.
- **Address Feedback:**
  - Incorporate feedback and make necessary adjustments.

6. Preparation for Implementation

- **Code Skeleton and Stubs:**
  - Create initial code structures and placeholders for components.
- **Implementation Planning:**
  - Plan the development tasks, timelines, and resource allocation.

The flow of software design, starting from the Software Requirements Specification (SRS) document,

7. Design Handoff and Transition

- **Handoff to Development Team:**
  - Ensure that developers have a clear understanding of the design and implementation plan.
- **Transition to Development Phase:**
  - Begin the coding and implementation based on the detailed design.

8. Continuous Design Evolution

- **Design Refinements:**
  - As development progresses, refine the design based on practical implementation considerations and new requirements.
- **Design Documentation Updates:**
  - Keep the design documents up to date with any changes or improvements.