Software Architecture

Indhold

Introduction:

- Software architecture refers to the high-level structure of a software system, encompassing the set of decisions about the organization of the system.
- It involves the design of software modules, their relationships, and how they interact with each other to fulfill system requirements.

- **Process**
  - The methodology and steps involved in creating, refining, and maintaining the architecture.
- **Documentation**
  - The ways in which the architecture is described and communicated.
- **Styles**
  - The architectural patterns and styles used to design the system.

Process

- **Definition**
  - The architecture process defines how the architecture is developed, maintained, and evolved over time.
- **Phases**
  - **Requirements Gathering**
    - Understanding the system's functional and non-functional requirements.
    - SRP (Single Responsibility Principle): Ensure each requirement addresses a single aspect of the system.
  - **Architecture Design**
    - Creating the initial high-level design of the system.
    - OCP (Open/Closed Principle): Design components to be open for extension but closed for modification.
  - **Implementation**
    - Translating the architectural design into code.
    - LSP (Liskov Substitution Principle): Ensure derived classes can be substituted for base classes.
  - **Verification**
    - Ensuring the architecture meets the requirements and functions as intended.
    - ISP (Interface Segregation Principle): Create specific interfaces that don't force clients to depend on methods they don't use.
  - **Maintenance**
    - Updating and refining the architecture as the system evolves.
    - DIP (Dependency Inversion Principle): Depend on abstractions, not on concretions, to ease maintenance and scalability.

Documentation

- **Importance**
  - Provides a blueprint for developers, helps in communication among stakeholders, and serves as a reference for future maintenance.
- **Types**
  - **Views**
    - Different perspectives of the system, such as logical, physical, development, and process views.
  - **Diagrams**
    - Visual representations, including class diagrams, sequence diagrams, component diagrams, and deployment diagrams.
  - **Specifications**
    - Detailed descriptions of components, interfaces, and interactions.
- **Best Practices**
  - **Clarity and Conciseness**
    - Ensure the documentation is easy to understand.
    - Apply SRP to documentation by ensuring each document focuses on a single aspect.
  - **Consistency**

    - Maintain uniformity in terminology and format.
    - Use OCP to allow documentation to be extended without major modifications.
- **Up-to-date**
    - Regularly update the documentation to reflect changes in the architecture.
    - Ensure LSP by making sure updates don't invalidate existing documentation.

Styles

- **Definition**

    - Architectural styles are predefined solutions to common architectural problems.

- **Examples**

    - **Layered Architecture**
        - Divides the system into layers with specific responsibilities.
        - SRP: Each layer has a single responsibility.
        - OCP: Layers can be extended with new functionalities without changing existing code.
        - LSP: Each layer can be replaced with another as long as the interface is maintained.
        - ISP: Each layer communicates with others through well-defined interfaces.
        - DIP: Layers depend on abstractions rather than concrete implementations.
    - **Client-Server Architecture**
        - Separates the client and server responsibilities.
    - **Microservices Architecture**
        - Composes the system of small, independent services.
    - **Event-Driven Architecture**
        - Uses events to trigger communication between decoupled services.

- **Layered Architecture Example**

    - **Definition**
        - Organizes the system into layers with distinct responsibilities, such as presentation, business logic, and data access.
    - **Benefits**
        - **Separation of Concerns**
            - Each layer handles specific aspects of the system, reducing complexity.
        - **Modularity**
            - Facilitates maintenance and testing.
        - **Scalability**
            - Easier to scale specific layers independently.