Purwadhika
Digital Technology School

Full Stack AI Software Development

Software Architecture, Design Patterns, and Software Development Lifecycle

Job Connector Program

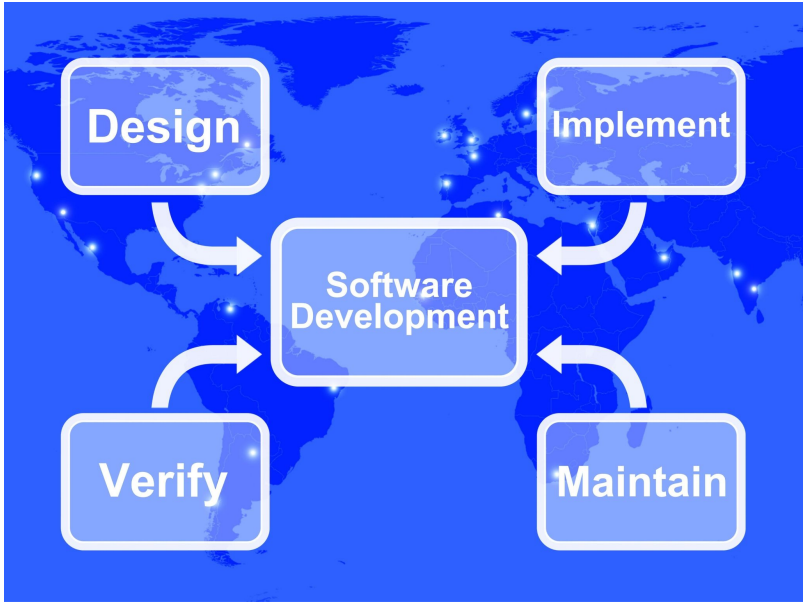
Outline

- Software Architecture
- Software Design Principles
- Software Design Patterns
- Software Development Lifecycle (SDLC)

What is Software Architecture?

Software architecture refers to the fundamental structure of a software system and the discipline of creating such structures and systems.

It encompasses various elements, including the organization of software components, their relationships, principles guiding their design, and the reasoning behind those decisions.



Importance of Software Architecture

Scalability

Allows a system to handle growing demands without compromising performance or functionality.

Real Case: A social media platform preparing for a major event (e.g., the Olympics) **adds hundreds of new servers to its infrastructure**. This scaling ensures the application can handle a 10x surge in simultaneous user posts and photo uploads without crashing or slowing down.

Maintainability

Facilitates easier maintenance and updates, reducing the cost and effort required to implement changes.

Real Case: A software team **uses a Microservices Architecture** where the billing service is separate from the shipping service. If they need to update tax laws in the billing code, they can update and deploy only the billing service without risking bugs in the shipping service or taking the entire application offline.

Importance of Software Architecture

Reliability

A robust structure minimizes vulnerabilities and failure points, enhancing the system's stability.

Real Case: An airline ticketing system is built to be fault-tolerant. **If the primary database server fails, a secondary server immediately takes over** (failover), ensuring users can complete their bookings and preventing lost sales.

Flexibility

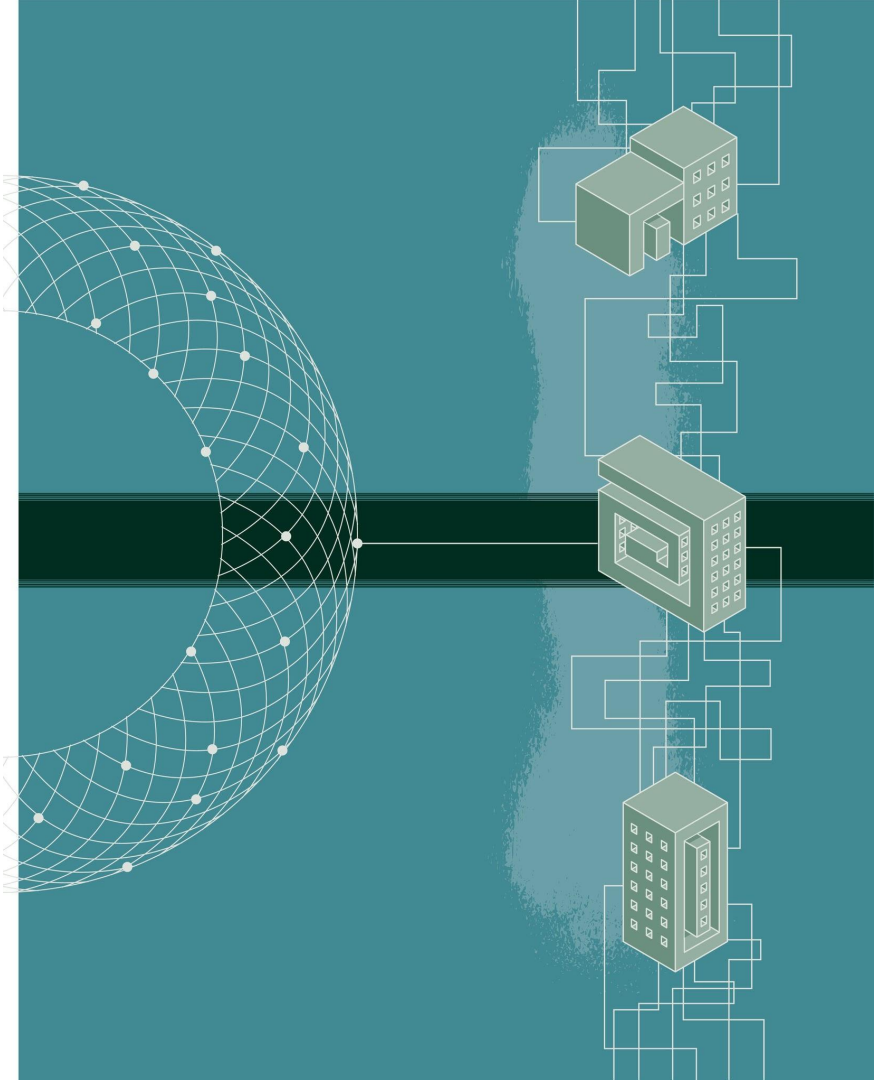
Enables the system to adapt to new requirements and technological advancements without significant rework.

Real Case: A company has an e-commerce website built with a Layered Architecture. When they decide to integrate a new third-party payment provider (e.g., a new mobile wallet), **the architecture's separation of concerns** allows them to update only the payment processing layer without rewriting the core business logic or user interface.

Key Architectural Styles

Different architectures offer distinct ways to structure a system, each with specific trade-offs:

- Monolithic Architecture
- Microservices Architecture
- Layered Architecture
- Event-Driven Architecture (EDA)
- Service-Oriented Architecture (SOA)



Monolithic Architecture

A traditional approach where all application components are tightly integrated and deployed as a single unit.

Advantages	Disadvantages
Simplicity: Easier to develop and test	**Scalability:** Difficult to scale horizontally
Deployment: Single deployment unit simplifies the process	**Technology Stack:** Limited flexibility in choosing technologies
Debugging: Easier to debug and trace issues (one codebase)	**Maintenance:** Changes may require redeployment of the entire application

Microservices Architecture

An approach where an application is divided into small, independent services that communicate through APIs. Each microservice handles a specific business capability.

Advantages	Disadvantages
Scalability: Easier to scale horizontally and independently	**Complexity:** Requires a robust infrastructure for managing and coordinating service
Technology Diversity: Different services can use different technologies	**Communication:** Inter-service communication adds complexity
Fault Isolation: Failure in one service doesn't necessarily affect the whole application	**Distributed Challenges:** Managing distributed data and transactions is difficult

Layered Architecture (N-Tier)

Organizes an application into layers, each with a specific set of responsibilities, such as presentation, business logic, and data storage.

Advantages	Disadvantages
Separation of Concerns: Clear separation of functionality	**Rigidity:** Changes in one layer may impact others, requiring modifications in multiple places
Maintainability: Easier to maintain as changes are localized	**Performance:** Inter-layer communication can introduce overhead
Reusability: Components within a layer can be reused	**Initial Development Time:** Designing and implementing this structure can be time-consuming

Event-Driven Architecture (EDA)

The flow is determined by events (e.g., user actions, messages). Components communicate asynchronously through these events.

Advantages	Disadvantages
Loose Coupling: Components are not dependent on each other, enhancing flexibility	**Complexity:** Implementing and managing event-driven systems can be complex
Scalability: Easier to scale as components react to events independently	**Debugging:** Asynchronous nature can make debugging challenging
Real-time Responsiveness: Well-suited for real-time applications	**Consistency:** Ensuring consistency in a distributed system can be difficult

Software Design Principles

These principles guide developers in making crucial decisions during the design phase to create robust, maintainable, and scalable software

DRY (Don't Repeat Yourself)

- Emphasizes the avoidance of redundancy in code.
- Promotes reusability by consolidating repeated code segments.
- Benefits: Reduces duplication, simplifies maintenance, and improves readability

KISS (Keep It Simple, Stupid)

- Encourages simplicity by advocating for straightforward solutions over overly complex ones.
- Benefits: Improves code readability, ease of maintenance, and reduces the chance of errors.

YAGNI (You Aren't Gonna Need It)

- Advises against adding functionality until deemed necessary.
- Discourages implementing features based on assumptions about future needs.
- Benefits: Prevents over-engineering, reduces unnecessary complexity, and maintains focus on immediate requirements.

SOLID Principles

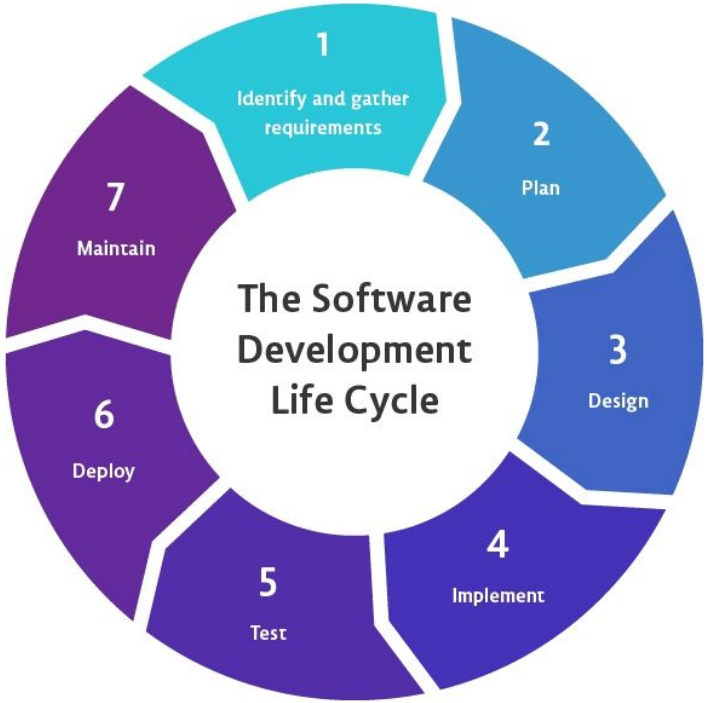
Acronym	Principle	Core Concept
S	**Single Responsibility Principle**	A class should have only **one reason to change**, meaning it should have only one job or responsibility
O	**Open/Closed Principle**	Software entities should be **open for extension but closed for modification**
L	**Liskov Substitution Principle**	Objects of a superclass should be **replaceable with objects of its subclasses** without affecting the program's functionality
I	**Interface Segregation Principle**	A client should **not be forced to depend on interfaces it does not use**
D	**Dependency Inversion Principle**	High-level modules should not depend on low-level modules; **both should depend on abstractions**

Overall Benefits: Enhances code maintainability, readability, and scalability while reducing coupling and increasing code reusability.

Software Development Lifecycle (SDLC)

A process used to design, develop, and test high-quality software.

- **Goal:** To produce software that meets or exceeds customer expectations and reaches completion within time and cost estimates.
- **Function:** Defines a detailed plan describing how to develop, maintain, replace, and enhance specific software.



SDLC Approaches

Approach	Heavyweight (Predictive)	Lightweight (Adaptive/Agile)
Scope	Scope of work is **predefined in advance**	Adaptive; a compromise between strict discipline and total absence
Documentation	Requires a **significant contribution** from programmers and **weighty documentation**	Requires a **much smaller amount** of documentation
Use Case	Traditionally chosen for **extensive projects**	Opens up new possibilities for flexibility and puts communication first
Examples	Waterfall, Iterative, Spiral, V-shaped	Scrum, Extreme Programming (XP), Kanban

Waterfall vs. Agile: Comparison

Feature	Waterfall	Agile
Approach	Goals and outcome established from the beginning	Frequent stakeholder interaction
Flexibility	Low	High
Workflow	**Linear system**; requires completing each phase before moving on to the next one	Encourages the team to **work simultaneously** on different phases
Progress Requires	Completing deliverables to progress to the next phase	Team initiative and short-term deadlines

Waterfall Model

Design Steps

- **Requirement Gathering and analysis:** Capture all requirements and document them.
- **System Design:** Study requirements and prepare the system design.
- **Implementation:** Develop the system in small programs (units).
- **Integration and Testing:** Integrate and test all units.
- **Deployment of system:** Deploy the product.
- **Maintenance:** Release patches to fix issues that come up in the client environment.

Use Cases

- When requirements are constant and not changed regularly.
- When the project is short, the situation is calm, and the technology used is consistent.
- When resources are well prepared and available to use.

Agile Model

Design Steps

- **Requirements gathering:** Define requirements, business opportunities, and plan effort.
- **Design the requirements:** Define requirements using diagrams like user flow or UML.
- **Construction/Iteration:** Designers and developers work to deploy a working product.
- **Testing:** QA team examines performance and looks for bugs.
- **Deployment:** The team issues a product for the user's work environment.
- **Feedback:** The team receives and works through feedback after release.

Use Cases

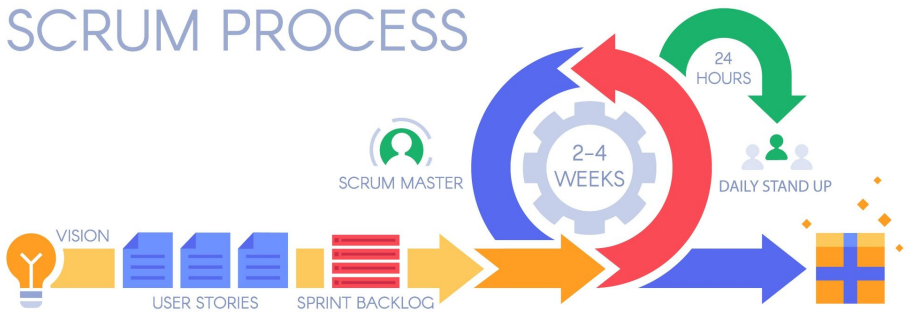
- When frequent changes are required.
- When a highly qualified and experienced team is available.
- When a customer is ready to have a meeting with the software team all the time.
- When project size is small.

Scrum (Agile Methodology)

An agile development process primarily focused on managing tasks in team-based development conditions.

Key Roles and Responsibilities:

- **Scrum Master:** Sets up the team, arranges meetings, and removes obstacles for the process.
- **Product Owner:** Manages the product backlog, prioritizes delays, and is responsible for the distribution of functionality on each iteration.
- **Scrum Team:** Manages its work and organizes the work to complete the sprint or cycle.



Project Management Tools

Project Management tools can be customized to fit the needs of various teams and goals.

Key Features:

- Planning/Scheduling: Tasks, subtasks, workflows, and calendars.
- Collaboration: Task assignment, commenting, and dashboards.
- Documentation: File editing, versioning, and storage.
- Evaluation: Track and assess productivity through reporting.
- Common Tools:
  - **Jira**
  - **Trello**
  - **Asana**

AI Role in Software Development

The integration of Artificial Intelligence (AI) into the Software Development Lifecycle (SDLC) is shifting the paradigm from manual coding to AI-Augmented Development, significantly increasing velocity and code quality.

Key Impact Areas:

Automated Code Generation

AI coding assistants (like GitHub Copilot or Cursor) use Large Language Models (LLMs) to provide real-time suggestions, boilerplate generation, and refactoring based on existing design patterns.

Enhanced Design & Architecture

AI tools can analyze complex requirements to suggest optimal Architectural Styles (e.g., Microservices vs. Monolith) by predicting potential scalability bottlenecks before a single line of code is written.

Thank You!