SOFTWARE DESIGN & ARCHITECTURE

Software Design

Software design is the process of defining how a software system will be built to meet specified requirements. It focuses on structure, components, interfaces, and data flow.

Software architecture

Software architecture is the high-level structure of a software system, describing:

- Major components
- Their responsibilities
- How they interact
- Design principles and constraints

• Good design reduces cost, risk, and failure in business systems

Difference Between Design and Architecture

• Architecture entails high-level structure of the system while design involve detailed implementation of the components
• Architecture focuses on big picture while design focuses on component behavior

Importance of Software Design in Business

1. **Improves scalability**
   A good software design allows the system to handle increased users, data, or workload without major changes, making it easier to expand as demand grows.

2. **Reduces maintenance cost**
   Well-structured and modular software is easier to understand, fix, and update, reducing time, effort, and cost spent on maintenance.

3. **Enhances performance and security**
   Proper architecture optimizes resource usage and enforces security controls, resulting in faster system response and better protection against threats.

4. **Supports business growth**
   Flexible and reliable software systems can adapt to new business requirements, integrate with new technologies, and support organizational expansion.

Software Design Principles

• Guide developers in making good design decisions
• Ensure maintainability and flexibility
• Reduce complexity

Key Principles

1. SOLID
2. DRY
3. KISS
4. Separation of Concerns

1. SOLID Principles

These are rules for building change-friendly business systems.

• S – Single Responsibility Principle
• O – Open/Closed Principle
• L – Liskov Substitution Principle
• I – Interface Segregation Principle
• D – Dependency Inversion Principle

2. DRY Principles

DRY – Don't Repeat Yourself

• Avoid duplicated logic

3. KISS principles

KISS– Keep It Simple, Stupid means Simplicity improves reliability

ARCHITECTURAL STYLES

An architectural style defines a pattern for organizing a software system, including:

- Types of components
- Relationships between components
- Rules and constraints for interaction

It provides a standard approach to designing systems.

• Define system organization
• Influence scalability and performance
• Must align with business needs

Layered Architecture

Layered architecture divides a software application into horizontal layers, with each layer focusing on a specific responsibility. The layers work together to process data, handle business logic, and interact with users, ensuring that no single layer bears the weight of all responsibilities.

Common Layers in Layered Architecture

Presentation Layer:

Handles the user interface and user interactions.

Examples: Web pages, mobile app interfaces, or desktop GUIs.

Business Layer:

Processes the application's business rules and workflows.

Examples: Validation logic, calculations, and core algorithms.

Persistence Layer:

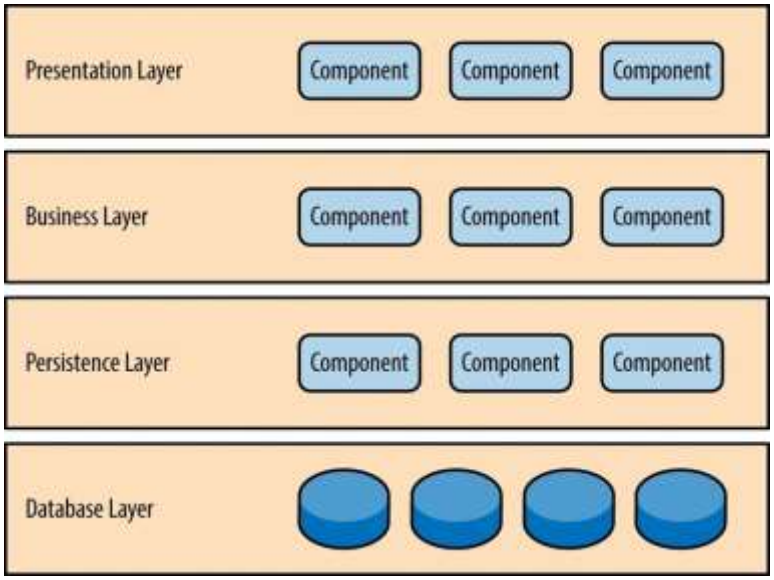
Manages data storage and retrieval.

Examples: Database queries, caching, and APIs.

Database Layer:

The actual data storage system where information is saved.

Examples: Relational databases (SQL Server, PostgreSQL) or NoSQL solutions (MongoDB).



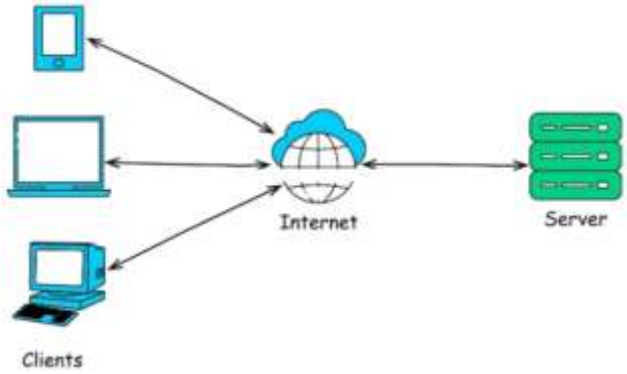
Client–Server

Key Components:

Client: The client is typically a device or application that initiates a request to the server. This could be a web browser, a mobile app, or a desktop application.

Server: The server is a powerful computer or software application that processes requests from clients, manages resources, and delivers the requested services or data.

Network: The communication medium (usually the internet) that allows clients and servers to exchange data.



How Client-Server Architecture Works

- How does your browser know what to show when you type in a URL?
- Or how does your Spotify app pull in your favorite playlist in seconds?

It all comes down to how clients and servers talk to each other.

Steps

1. The Client Initiates a Request: You (the client) perform an action like clicking a link, pressing “Send” on an email, or opening an app. That action triggers a request to a server.
2. The Request Travels Over the Network: This request usually in the form of an HTTP message is sent over the internet to a server’s IP address. Think of it like mailing a letter to a specific address.
3. The Server Receives and Processes the Request: The server listens on a specific port and handles incoming requests. It processes the data, runs logic, queries a database if needed, and prepares a response.
4. The Server Sends Back a Response: Once processing is done, the server sends the result back. This could be: A webpage, Search results, A confirmation message, JSON data for a mobile app

5. The Client Displays the Response: The client receives the response and renders it on screen. What you see in your browser or app is the result of this back-and-forth.

Key Technologies Involved

Here are some of the technologies that enable this communication:

- HTTP/HTTPS: The most common protocol used by browsers and web servers for communication.
- DNS (Domain Name System): Translates human-friendly domain names (like algomaster.io) into server IP addresses.
- TCP/IP: The underlying protocol that ensures data packets are delivered reliably between client and server.
- Ports: Servers listen on specific ports (like 80 for HTTP or 443 for HTTPS) to accept requests.

Advantages of Client-Server Architecture

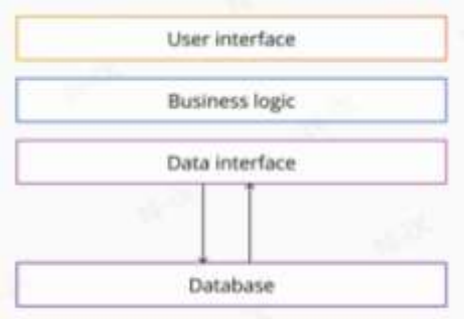
The client-server model offers several advantages, which is why it's so widely used:

a) Centralized Management: All critical data, resources, and services reside on the server, which means they can be monitored, updated, and secured from a central location.
b) Scalability: Client-server systems are built to grow. You can scale vertically by upgrading server hardware (e.g., adding more RAM or CPU) or scale horizontally by adding more servers behind load balancers to distribute traffic.
c) Efficient Resource Sharing: One server can serve multiple clients simultaneously, enabling shared access to databases, file storage, and applications.
d) Security: With everything centralized, it's easier to implement and enforce authentication, authorization, encryption, and access control.

Monolithic Architecture

Monolithic architecture is a software design methodology that combines all of an application's components into a single, inseparable unit. Under this architecture, the user interface, business logic, and data access layers are all created, put into use, and maintained as one, unified unit.

- A traditional approach in system design, which contains all application components into a single codebase.
- It was preferred for its simplicity and ease of initial setup.
- In contrast, alternative architectural approaches, like microservices, divide the application into smaller, separately deployable services.
- Because of its rigidity, it is difficult to scale and maintain, which makes it difficult to adjust to changing needs.



Design Principles of Monolithic Systems

Monolithic system design focuses on preserving manageability, consistency, and simplicity within a single codebase. Some of the key design principles are:

a) Modularity: Even though a monolithic system consists of a single codebase, it's essential to structure the code in a modular way.
b) Separation of Concerns: According to the separation of concerns principle, several application components should be responsible for separate tasks. For instance, debugging is made easier and code organization is made clearer by separating the user interface logic from business and data access logic.
c) Scalability: Architecting the system to support horizontal scaling when necessary is known as "scalability design." This might involve introducing asynchronous processing for resource-intensive operations, employing caching methods, or optimizing performance-critical components.
d) Encapsulation: Encapsulation is the process of revealing only the interfaces that are required for interaction while hiding the core operations of a component. Developers can reduce dependencies and make code maintenance and evolution easier by encapsulating functionality within clearly defined interfaces.
e) Consistency: Maintaining consistency in coding styles, architectural patterns, and design principles across the entire codebase ensures clarity and predictability for developers.

Advantages of monolithic

- Monolithic architecture is simple and easy to understand as it involves building the entire application in a single codebase with one set of languages and frameworks.
- Monolithic architecture typically performs better than distributed architecture as it does not require network communication between different components.
- Testing monolithic applications is easier as there is only one codebase to test. This way, you can easily ensure that all components work together correctly.
- Monolithic applications are easier to deploy as you only need to deploy a single codebase rather than deploying multiple components separately.

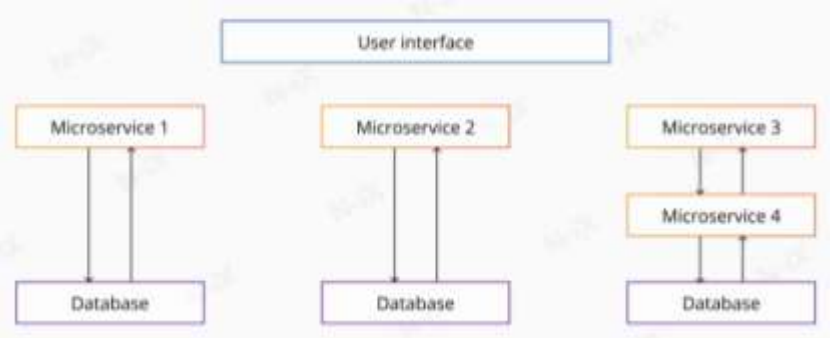
- Monolithic architecture is generally less expensive to develop and maintain as it doesn't require various technologies and tools, reducing the complexity of the application.
- With monolithic architecture, ensuring that security measures are applied consistently across the entire application is more accessible.

Disadvantages of the Monolithic Architecture

- Updating to any part of the codebase impacts the monolithic architecture, requiring complete app recompilation.
- Any errors or server issues impact the entire application, impacting its overall reliability.
- Code reuse is limited, often only supported with shared libraries (leading to coupling issues).
- Changes to any part of the app code become expensive due to dependencies.
- The code base can become significant over time, marking it challenging to maintain and for new developers to contribute.
- In monolithic architecture, you're tied to a single technology stack for everything.

Microservices

Microservices architecture decouples the front end and back end of architecture, linking various independent services in the back end (microservices) to the front end via API. The microservices approach supports the flexibility to choose and scale services as needed.



Principles of Microservices

The correct Microservice principles can help developers build an application that performs and scales easily. The microservice architecture operates on the following principles:

1. Service Independence: Each microservice should be independent of other microservices, with its database and business logic.
2. Decentralization: Microservices architecture should be decentralized, with each service's team responsible for its development, deployment, and maintenance.

3. Agility: Microservices architecture should enable fast and flexible development, deployment, and scaling of individual services without affecting the entire system.
4. Resilience: Microservices should be fault-tolerant, with each service able to handle errors and failures independently.
5. Scalability: Microservices architecture should enable easy scaling of individual services to meet changing demands without affecting the entire system.

Advantages of the Microservices Architecture

1. Since you can scale each service independently of others, organizations can handle high traffic and spikes in usage more efficiently in the microservice architecture.
2. Organizations can quickly and easily add new features and services in the microservices architecture, making adapting to changing business requirements easier.
3. Microservices architecture is resilient to failures, as each service operates independently of others, reducing the risk of cascading losses and enabling faster recovery.
4. Maintaining the microservices architecture is more accessible than monolithic architecture, as each service is separate and independent, making modifying, updating, and testing individual services easier.
5. Microservices architecture is flexible, allowing organizations to use different technologies and programming languages for each service based on their specific needs.

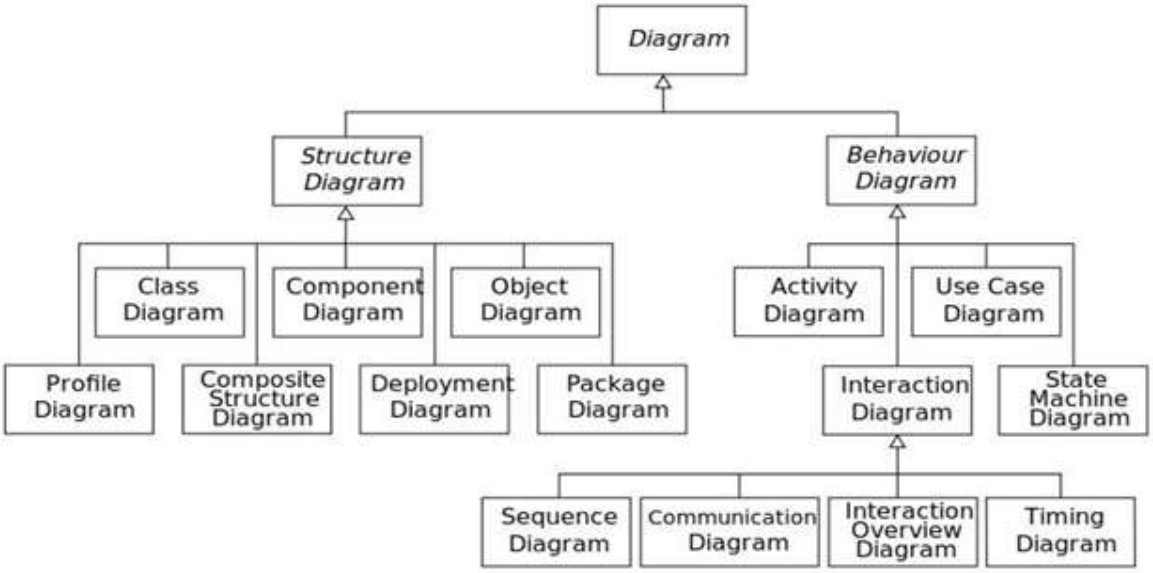
Disadvantages of Microservices Architecture

1. A microservice architecture can be more complex to design, implement, and maintain than a monolithic architecture. With a more significant number of services, there are more components to manage, making it harder to understand the system as a whole.
2. Since microservices are distributed systems, they can suffer from latency, network failures, and synchronization problems.
3. With more services to manage, the Microservice architecture has an increased operational overhead. It can include service discovery, load balancing, and deployment tasks.
4. In a microservice architecture, services need to communicate with each other over a network, which can add communication overhead.
5. Testing microservices can be more complex than testing a monolithic architecture. With more services to experiment with, there are more integration points and scenarios to consider.

UML DIAGRAM

Introduction

UML is a modeling language used in software engineering. It is very popular among OOA, OOD, OOP. UML was developed in the 1990s and adapted as an ISO standard in 2000. UML 2.2 has 14 different types of diagrams.



We have 2 main categories of diagrams namely Structure Diagrams and Behavior Diagrams

Why use UML?

1. **Design:**
   - Forward Design: doing UML before coding. Makes it easier to create the code in a structured manner

o Backward Design: doing UML after coding as documentation

2. **Code**
   - Some tools can auto-generate Code from UML diagrams

UML Software

There exist hundreds of different software for creating UML diagrams e.g:

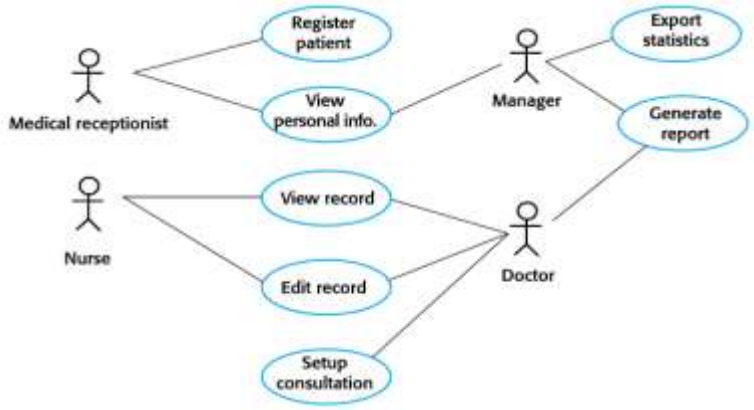
- Enterprise Architect
- StarUML
- Diagram tools like Lucidchart, Miro, Draw.io, etc.

Some of these software tools are free to use while others cost money.

Use Case Diagram

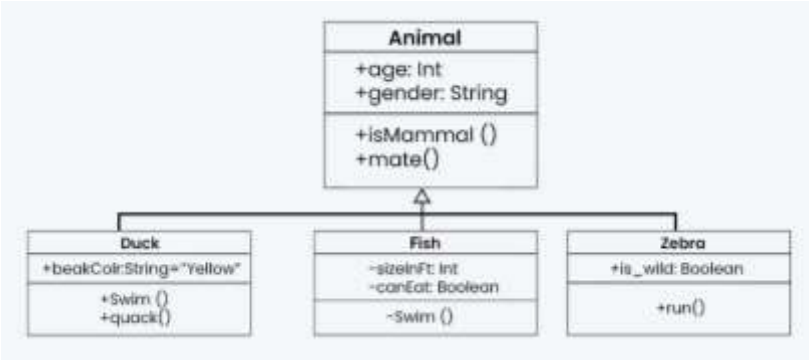
Use Case Diagrams are used to depict the functionality of a system or a part of a system. They are widely used to illustrate the functional requirements of the system and its interaction with external agents(actors).

A use case is basically a diagram representing different scenarios where the system can be used. A use case diagram gives us a high level view of what the system or a part of the system does without going into implementation details



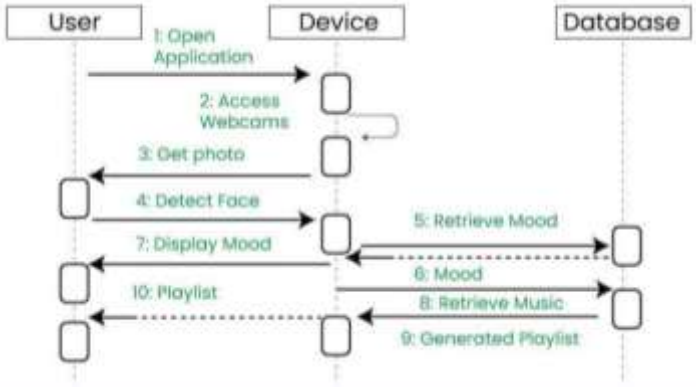
Class Diagram

The most widely use UML diagram is the class diagram. It is the building block of all object oriented software systems. We use class diagrams to depict the static structure of a system by showing system's classes, their methods and attributes. Class diagrams also help us identify relationship between different classes or objects.



Sequence Diagram

A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place. We can also use the terms event diagrams or event scenarios to refer to a sequence diagram. Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.



Activity Diagram

We use Activity Diagrams to illustrate the flow of control in a system. We can also use an activity diagram to refer to the steps involved in the execution of a use case. We model sequential and concurrent activities using activity diagrams. So, we basically depict workflows visually using an activity diagram. An activity diagram focuses on condition of flow and the sequence in which it happens. We describe what causes a particular event using an activity diagram