Architectural Patterns

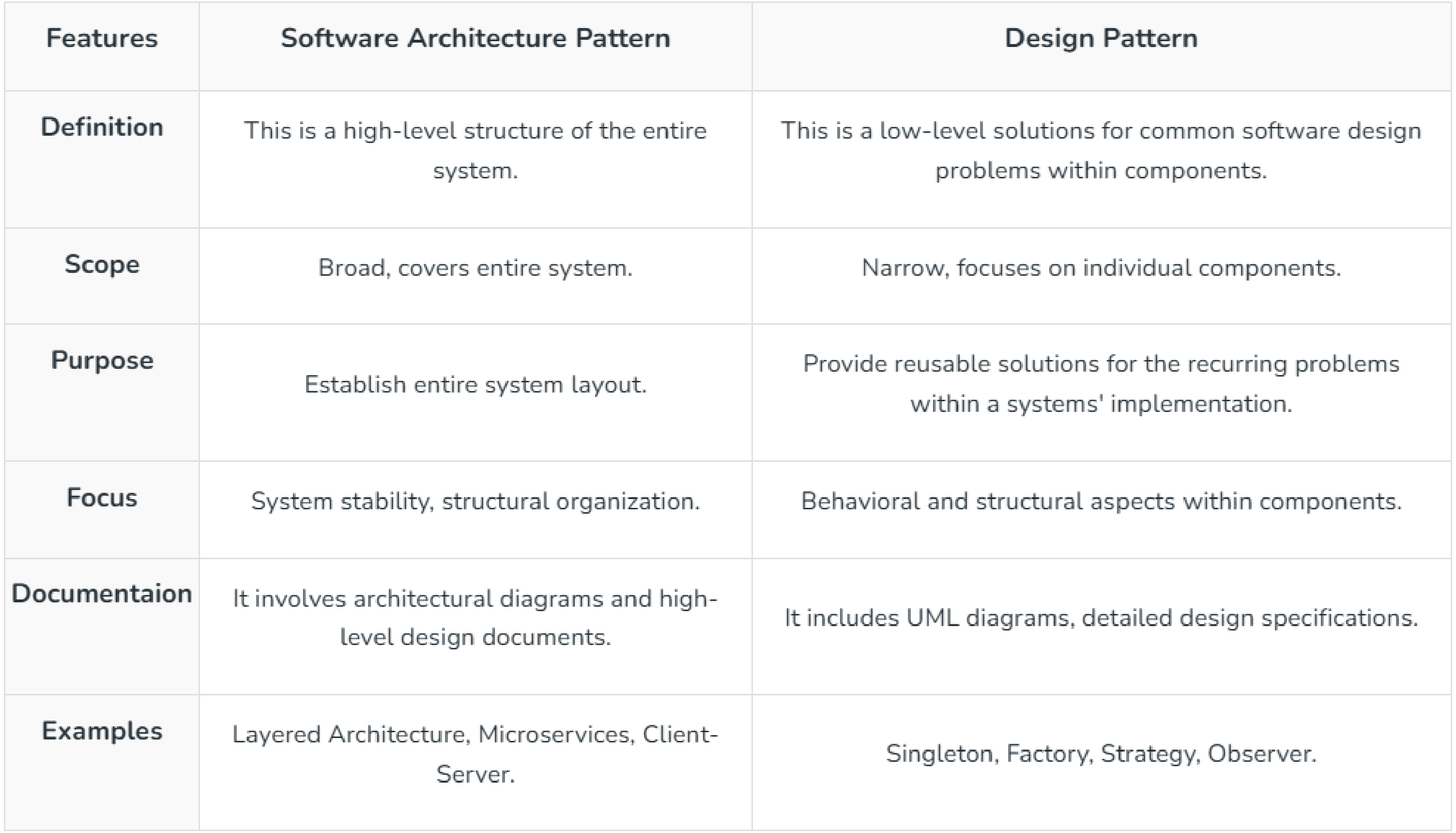
What is Software Architecture

Software Architecture is a high-level structure of the software system that includes the set of rules, patterns, and guidelines that dictate the organization, interactions, and the component relationships. It serves as blueprint ensuring that the system meets its requirements and it is maintainable and scalable.

Benefits of a Good Architectural Design

1. **Modularity**
2. **Encapsulation**
3. **Security**
4. **Documentation**
5. **Performance**

Architectural Pattern vs Design Pattern



Features	Software Architecture Pattern	Design Pattern
Definition	This is a high-level structure of the entire system.	This is a low-level solutions for common software design problems within components.
Scope	Broad, covers entire system.	Narrow, focuses on individual components.
Purpose	Establish entire system layout.	Provide reusable solutions for the recurring problems within a systems' implementation.
Focus	System stability, structural organization.	Behavioral and structural aspects within components.
Documentaion	It involves architectural diagrams and high-level design documents.	It includes UML diagrams, detailed design specifications.
Examples	Layered Architecture, Microservices, Client-Server.	Singleton, Factory, Strategy, Observer.

Layered Architecture Pattern

Layered Architecture Pattern

Layered architecture patterns are n-tiered patterns where the components are organized in horizontal layers. This is the traditional method for designing most software and is meant to be self-independent. This means that all the components are interconnected but do not depend on each other.

Example:

In an online shop, the **UI layer** shows pages, the **application layer** handles requests, the **business layer** checks rules like discounts, and the **data layer** reads and saves product and order information in the database.

Layered Architecture Pattern

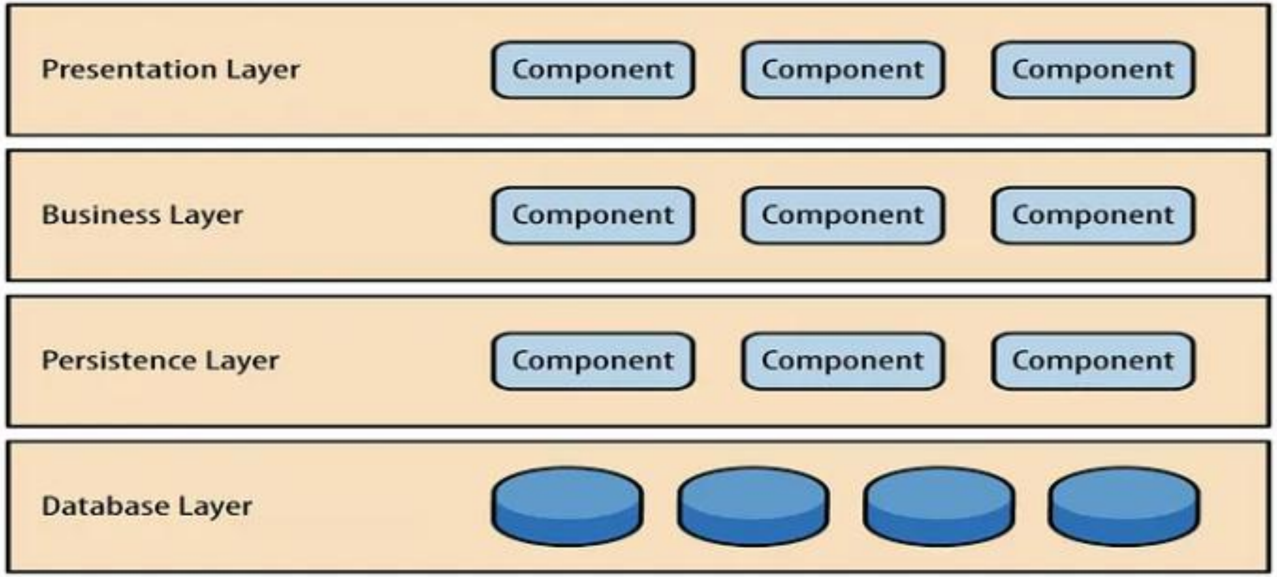


Image 1: Layered Architecture

Layered Architecture Pattern

1. **Presentation layer:** The user interface layer where we see and enter data into an application.
2. **Business layer:** This layer is responsible for executing business logic as per the request.
3. **Persistence/Application layer:** This layer acts as a medium for communication between the 'presentation layer' and 'data layer'. It’s used for handling functions like object-relational mapping
4. **Data Layer:** This layer has a data storage systems or database(s) for managing data.

What is Business Logic

Business logic is the part of a software system that encodes **the actual rules of the business**: how things *should* work according to domain rules, not according to technical concerns like databases, UI, or networking.

Pros and Cons of Layered Architecture

Pros:

1. **Scalability:** Individual layers in the architecture can be scaled independently to meet performance needs.
2. **Flexibility:** Different technologies can be used within each layer without affecting others.
3. **Maintainability:** Changes in one layer do not necessarily impact other layers, thus simplifying the maintenance.

Cons:

1. **Complexity:** Adding more layers to the architecture can make the system more complex and difficult to manage.
2. **Performance Overhead:** Multiple layers can introduce latency due to additional communication between the layers.
3. **Strict Layer Separation:** Strict layer separation can sometimes lead to inefficiencies and increased development effort.

Client-Server Architecture

Client-Server Architecture Pattern

Client-server architecture is a network model in which two main entities — clients and servers — communicate with each other to complete specific tasks or share data.

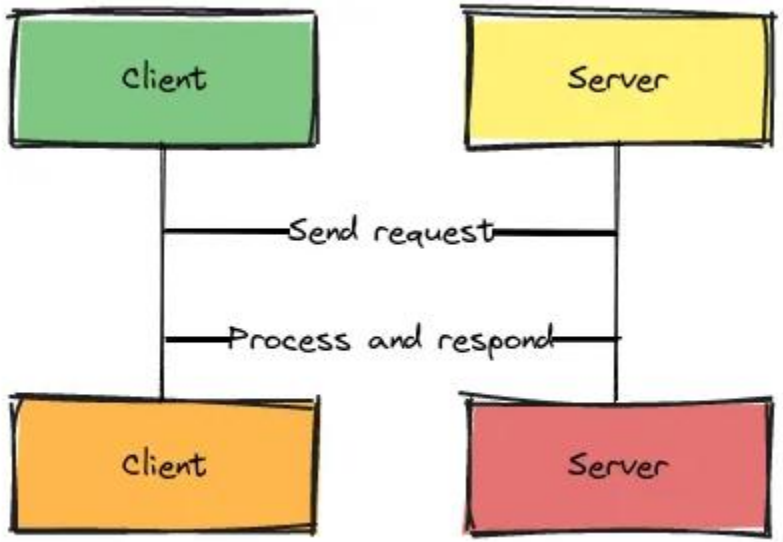
Client: The client initiates requests, waits for the server's response, and displays it to the user. For example, a web browser acts as a client when it requests a website.

Server: The server processes these requests, retrieves the relevant information, and sends it back to the client. For example, a web server responds with website data when a browser requests it.

Example

Client–server architecture is when one computer (client) asks another computer (server) for services. For example, on your phone you open a banking app **(client)**; it sends a request over the internet to the bank’s **server**, which checks your balance and sends the result back for you to see and use.

Client-Server Architecture Pattern



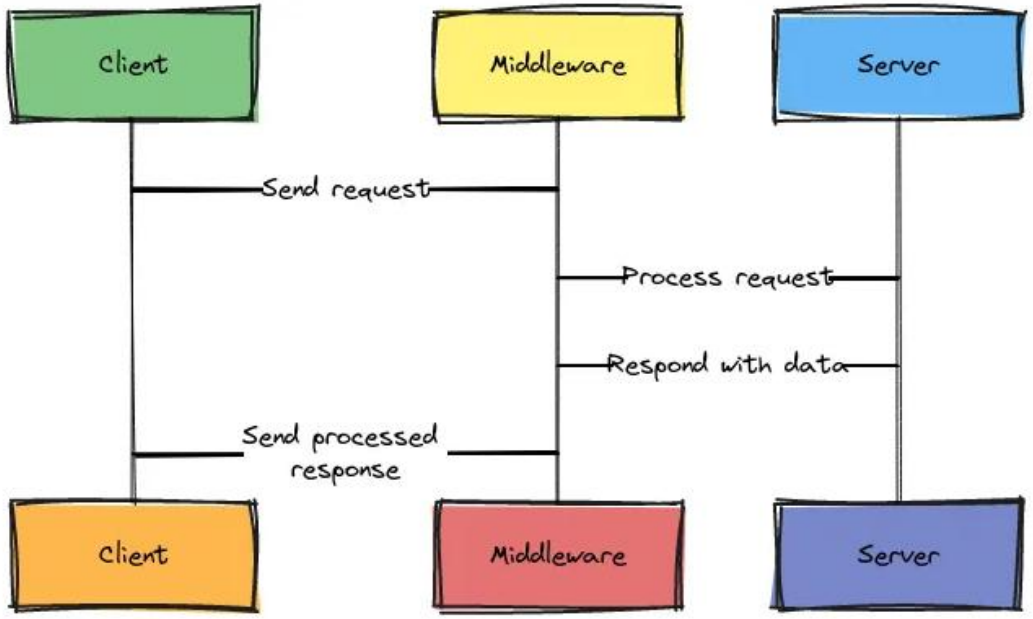
Two tier client server architecture

Types of Client-Server Architecture

Types of Client-Server Architecture

1. **Two-Tier Architecture**: The simplest form where clients and servers communicate directly.
2. **Three-Tier Architecture**: A middleware layer, often called the application layer, is added between the client and server.
3. **N-Tier Architecture**: Involves multiple intermediary layers, such as security and business logic, to manage complex data processing and ensure security.

3-Tier Client-Server Architecture



Three tier client server architecture

Pros and Cons of Client-Server Architecture

Pros of Client-Server Architecture

1. **Centralized Control**: Servers can handle multiple clients simultaneously, allowing for centralized management of resources and data.
2. **Scalability**: The architecture is scalable, allowing developers to add more clients or servers as needed without affecting the entire system.
3. **Data Security**: Servers can be secured with layers of security controls, protecting sensitive information and user data.

Pros and Cons of Client-Server Architecture

Cons of Client-Server Architecture

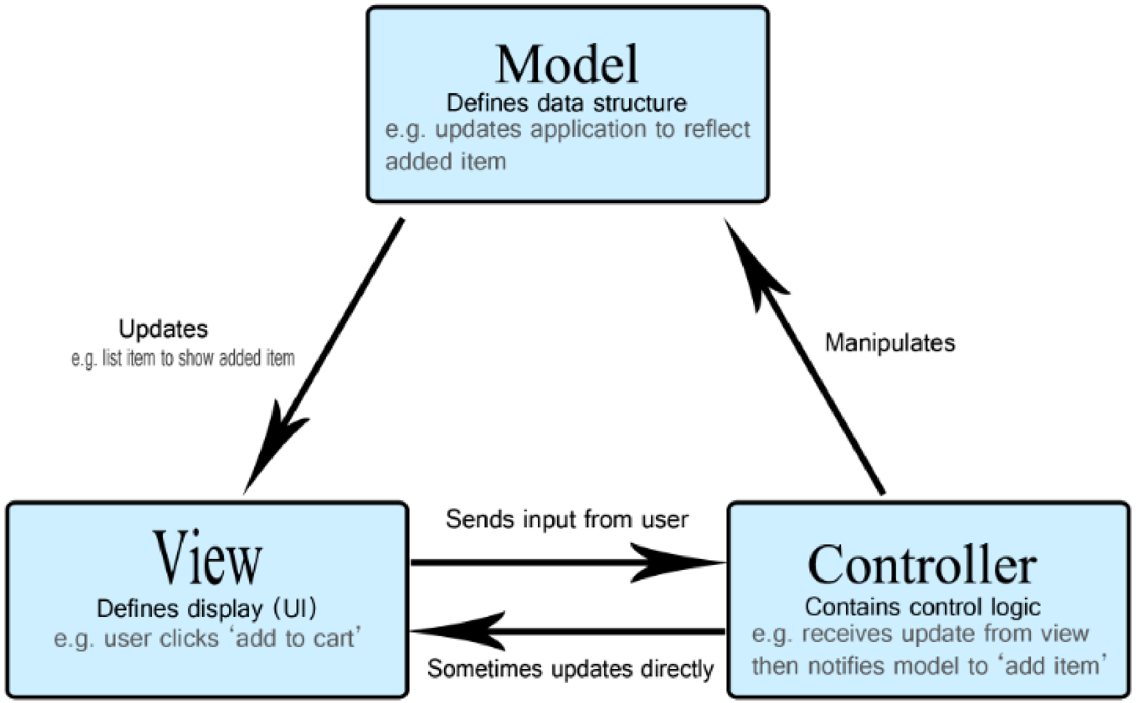
1. **Single Point of Failure:** If the server goes down, all connected clients lose access to data and services.
2. **Network Dependency:** Since the model relies on network communication, poor connectivity can lead to performance issues.
3. **Resource-Intensive:** Servers require significant resources to manage and serve multiple clients, which may increase infrastructure costs.

MVC Architecture

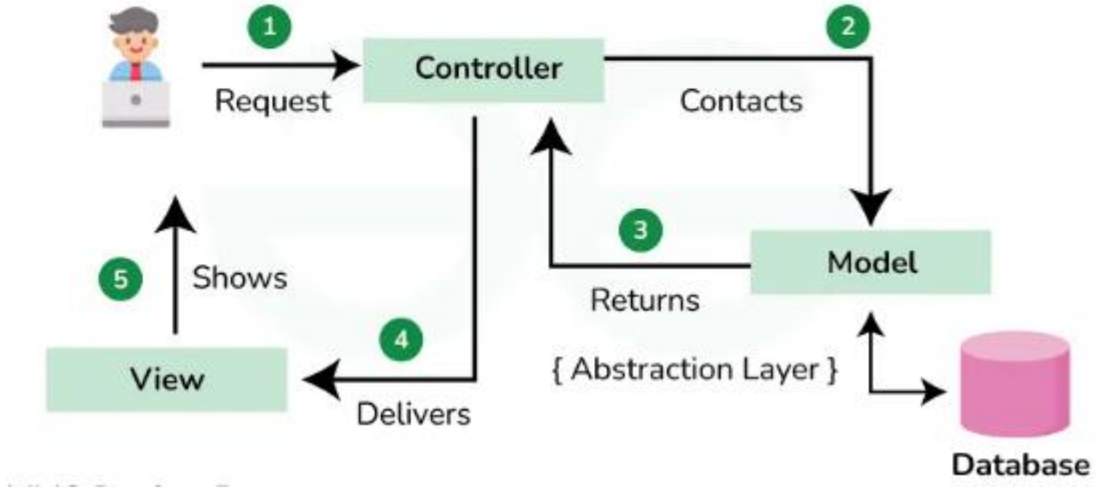
MVC Architectural Pattern

The **MVC pattern** is a software architecture pattern that separates an application into three main components: Model, View, and Controller, making it easier to manage and maintain the codebase. It also allows for the reusability of components and promotes a more modular approach to software development.

MVC Architectural Pattern


Model
Defines data structure
e.g. updates application to reflect added item
Updates
e.g. list item to show added item
Manipulates
View
Defines display (UI)
e.g. user clicks 'add to cart'
Sends input from user
Sometimes updates directly
Controller
Contains control logic
e.g. receives update from view then notifies model to 'add item'

MVC Architectural Pattern



MVC Architectural Pattern

The Model:

The model defines what data the app should contain. If the state of this data changes, then the model will usually notify the view (so the display can change as needed) and sometimes the controller (if different logic is needed to control the updated view).

The View:

The view defines how the app's data should be displayed. Generally, the UI functionality of a software application.

The Controller:

The controller contains logic that updates the model and/or view in response to input from the users of the app.

A Real-life Scenario

Imagine a simple shopping list app. All we want is a list of the name, quantity and price of each item we need to buy this week. Below we'll describe how we could implement some of this functionality using MVC.

A Real-life Scenario

The Model:

Going back to our shopping list app, the model would specify what data the list items should contain — item, price, etc. — and what list items are already present.

The View:

In our shopping list app, the view would define how the list is presented to the user, and receive the data to display from the model.

The Controller:

So for example, our shopping list could have input forms and buttons that allow us to add or delete items. These actions require the model to be updated, so the input is sent to the controller, which then manipulates the model as appropriate, which then sends updated data to the view.

When to Use the MVC Design Pattern

- **Complex Applications**: Use MVC for apps with many features and user interactions, like e-commerce sites. It helps organize code and manage complexity.
- **Frequent UI Changes**: If the UI needs regular updates, MVC allows changes to the View without affecting the underlying logic.
- **Reusability of Components**: If you want to reuse parts of your app in other projects, MVC's modular structure makes this easier.
- **Testing Requirements**: MVC supports thorough testing, allowing you to test each component separately for better quality control.

When Not to Use the MVC Design Pattern

- **Simple Applications**: For small apps with limited functionality, MVC can add unnecessary complexity. A simpler approach may be better.
- **Real-Time Applications**: MVC may not work well for apps that require immediate updates, like online games or chat apps.
- **Tightly Coupled UI and Logic**: If the UI and business logic are closely linked, MVC might complicate things further.
- **Limited Resources**: For small teams or those unfamiliar with MVC, simpler designs can lead to faster development and fewer issues.

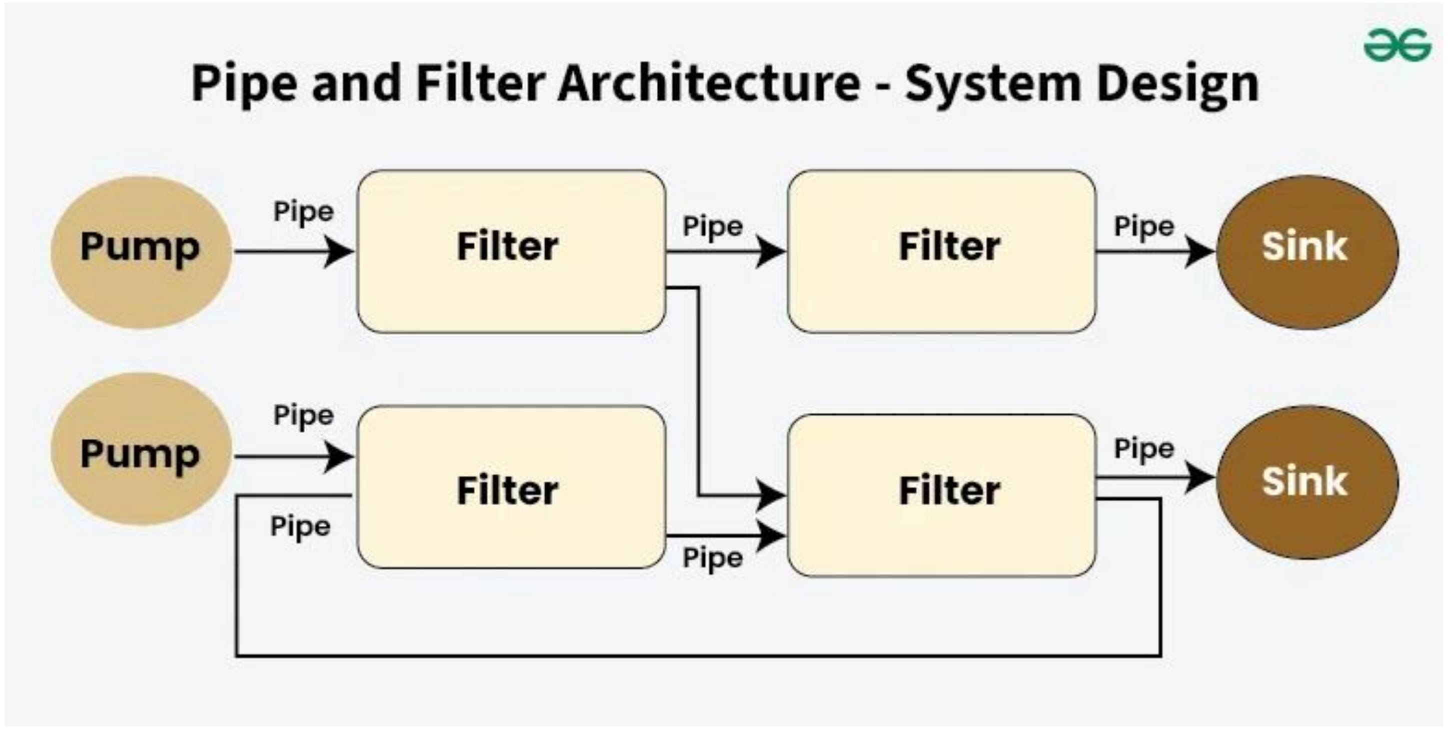
Pipe and Filter Architecture

Pipe and Filter Architecture

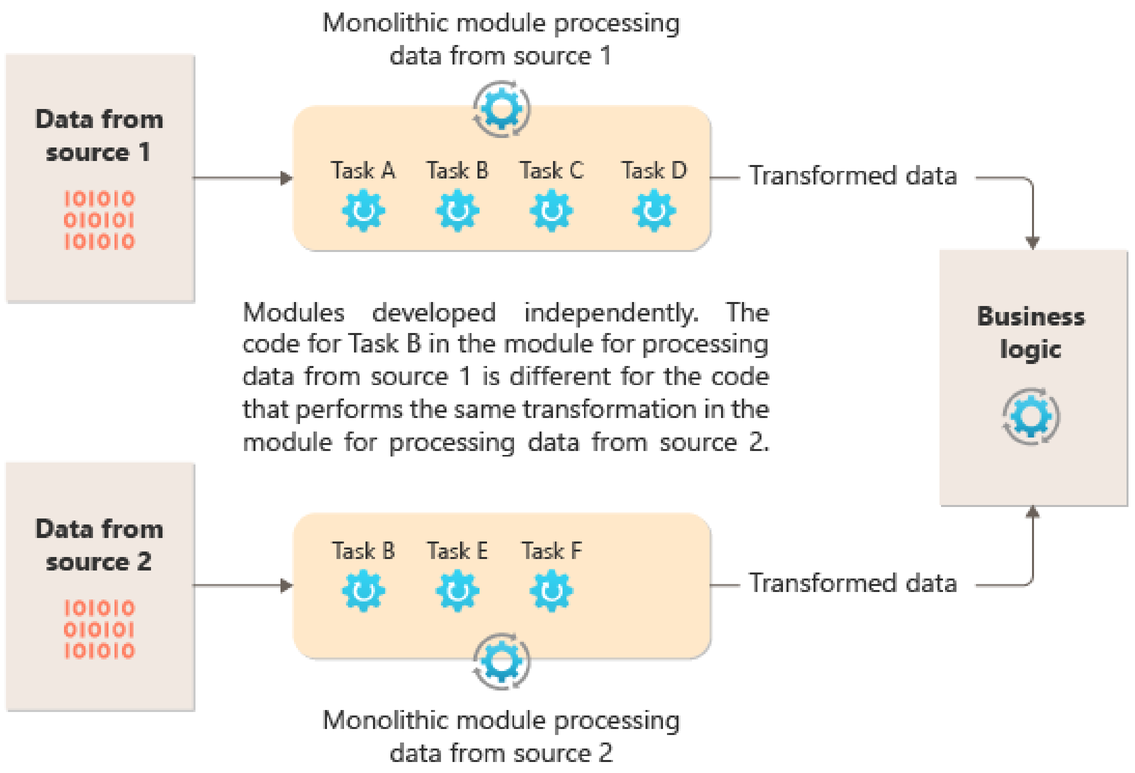
Pipe and Filter architecture structures processing tasks into a sequence of stages known as "pipes," where each stage filters and transforms data incrementally. This modular framework enables filters to operate independently, improving scalability and reusability.

- Each filter is tasked with specific operations, such as validating or formatting data, and passes its output to the next stage via interconnected pipes.
- This architecture ensures flexibility and ease of maintenance by isolating concerns regarding data analysis, allowing components to be reused across various systems.

Pipe and Filter Architecture



Pipe and Filter Architecture


Monolithic module processing data from source 1
Data from source 1
101010 010101 101010
Task A
Task B
Task C
Task D
Transformed data
Modules developed independently. The code for Task B in the module for processing data from source 1 is different for the code that performs the same transformation in the module for processing data from source 2.
Business logic
Data from source 2
101010 010101 101010
Task B
Task E
Task F
Transformed data
Monolithic module processing data from source 2

Pipe and Filter Architecture

Common use-case Scenario

The Pipe and Filter architecture is a versatile design pattern that can be applied in various domains and applications. Here are some common use cases and applications for this architecture:

- **Data Processing Pipelines**
    - **Text Processing:** Unix pipelines (e.g., grep, awk, sed) allow chaining commands to process and transform text data efficiently.
    - **Compilers:** Use a series of filters for lexical analysis, syntax parsing, semantic analysis, optimization, and code generation.

Pipe and Filter Architecture

Common use-case Scenario

The Pipe and Filter architecture is a versatile design pattern that can be applied in various domains and applications. Here are some common use cases and applications for this architecture:

- **Data Processing Pipelines**
  - **Text Processing:** Unix pipelines (e.g., grep, awk, sed) allow chaining commands to process and transform text data efficiently.
  - **Compilers:** Use a series of filters for lexical analysis, syntax parsing, semantic analysis, optimization, and code generation.

Pipe and Filter Architecture

Benefits of Pipe and Filter Architecture

- Enhanced Data Processing
- Ease of Understanding
- Isolation of Faults
- Improved Testing
- Standardization
- Resource Optimization

Pipe and Filter Architecture

Challenges of Pipe and Filter Architecture

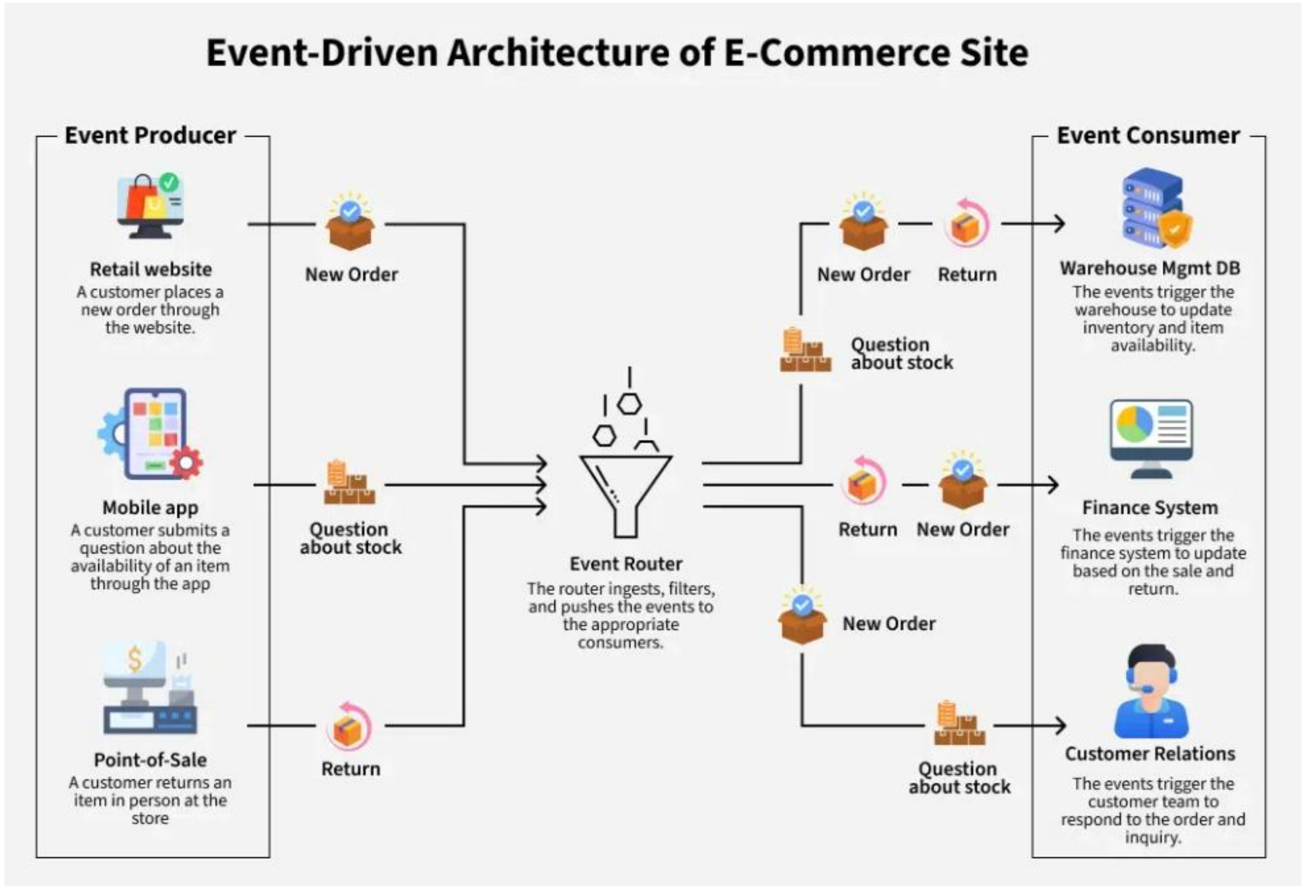
- **Performance Overhead**
- **Latency**
- **Complex Error Handling**
- **State Management**
- **Resource Utilization**

Event-driven Architecture

Event-driven Architecture

Event-Driven Architecture (EDA) is a software design paradigm where system components communicate by producing and responding to events. These events can be important happenings, like user actions or changes in the system's state. Components are independent, meaning they can function without being tightly linked to one another. When an event takes place, a message is dispatched, prompting the relevant components to respond accordingly.

A Real-life Scenario


Event-Driven Architecture of E-Commerce Site
Event Producer
Retail website
A customer places a new order through the website.
Mobile app
A customer submits a question about the availability of an item through the app
Point-of-Sale
A customer returns an item in person at the store
New Order
Question about stock
Return
Event Router
The router ingests, filters, and pushes the events to the appropriate consumers.
New Order
Return
Question about stock
Return
New Order
New Order
Question about stock
Event Consumer
Warehouse Mgmt DB
The events trigger the warehouse to update inventory and item availability.
Finance System
The events trigger the finance system to update based on the sale and return.
Customer Relations
The events trigger the customer team to respond to the order and inquiry.

Components of Event-Driven Architecture(EDA)

Event Source: This refers to any system or component that generates events. Examples include user interfaces, sensors, databases, and external systems.

Event: The core unit of communication in EDA, representing significant occurrences or changes in state.

Event Broker/Event Bus: Acting as a central hub, the event broker or event bus facilitates communication between various components by handling event distribution, filtering, and routing. It ensures that events reach the right subscribers, promoting efficient interaction within the system.

Publisher: This component generates and sends events to the event bus.

Subscriber: A component that shows interest in particular event types and subscribes to them. Subscribers listen for relevant events on the event bus and take action accordingly.

Components of Event-Driven Architecture(EDA)

Event Handler: This is a piece of code or logic linked to a subscriber that defines how to process received events.

Dispatcher: In some EDA implementations, a dispatcher is used to route events to the appropriate event handlers.

Aggregator: Several related events are combined by an aggregator to create a single, more significant event.

Listener: A component that actively monitors the event bus for events and reacts to them.

When to Use Event-Driven Architecture

- **Real-Time Applications**: If your application needs to react instantly to user actions or changes in data, EDA can provide the **responsiveness** required.
- **Scalability Needs**: When you expect your system to grow and handle an increasing number of events, EDA allows for better **scalability**. Components can be added or modified without disrupting the whole system.
- **Decoupled Components**: If you want to promote a **modular design**, EDA helps by allowing components to communicate through events rather than direct calls. This makes it easier to update or replace parts of your system.
- **Complex Event Processing**: For applications that need to handle multiple events and derive insights from them, EDA can simplify the **processing** of these complex scenarios.
- **Integration of Diverse Systems**: If you're working with various systems or services that need to communicate, EDA can help integrate them more effectively, enabling **flexible communication** between different technologies.

Challenges of Event-Driven Architecture(EDA)

- **Increased Complexity**: As more events and components are added, EDA systems can get complicated. It can be tough to manage how events flow and to keep everything coordinated.
- **Event Order and Consistency**: Keeping events in the right order and making sure the system remains consistent can be tricky. Handling events that come in out of sequence or ensuring that actions are completed as a group can require extra effort.
- **Debugging and Tracing**: Finding and fixing issues in a distributed and asynchronous setup can be harder than in traditional systems. It might take more time to track down problems.
- **Event Latency**: Because events are processed individually, there can be delays between when an event occurs and when it gets responded to. This lag might be an issue in situations that require quick reactions.

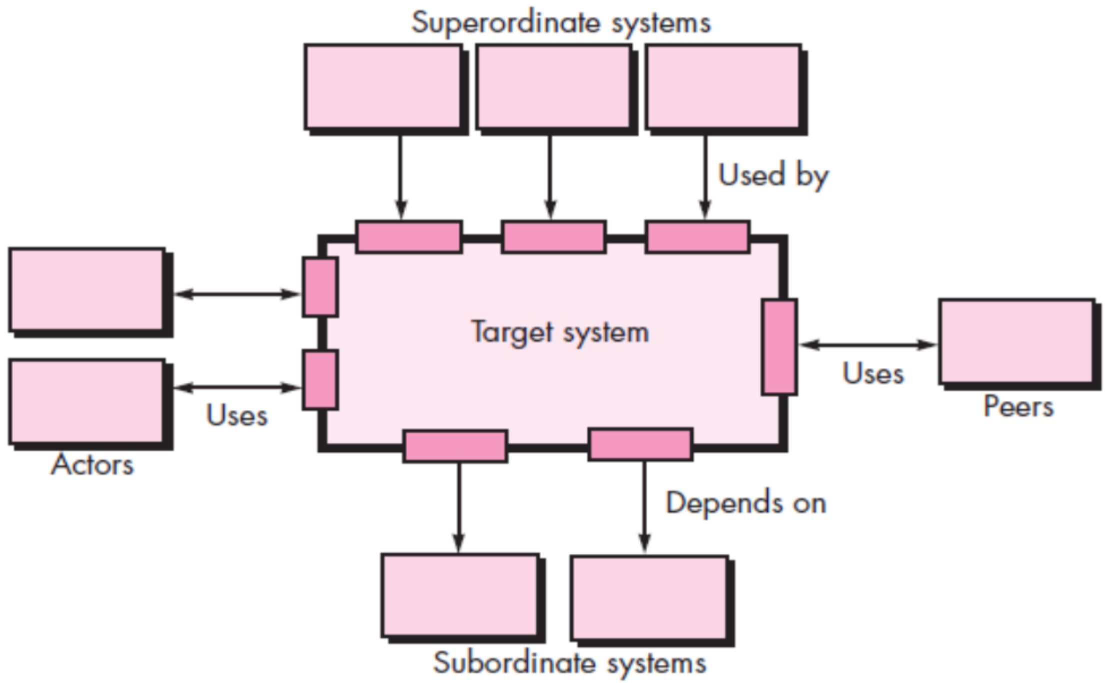
Designing Initial Architectural Components

Architectural Context Diagram (ACD)

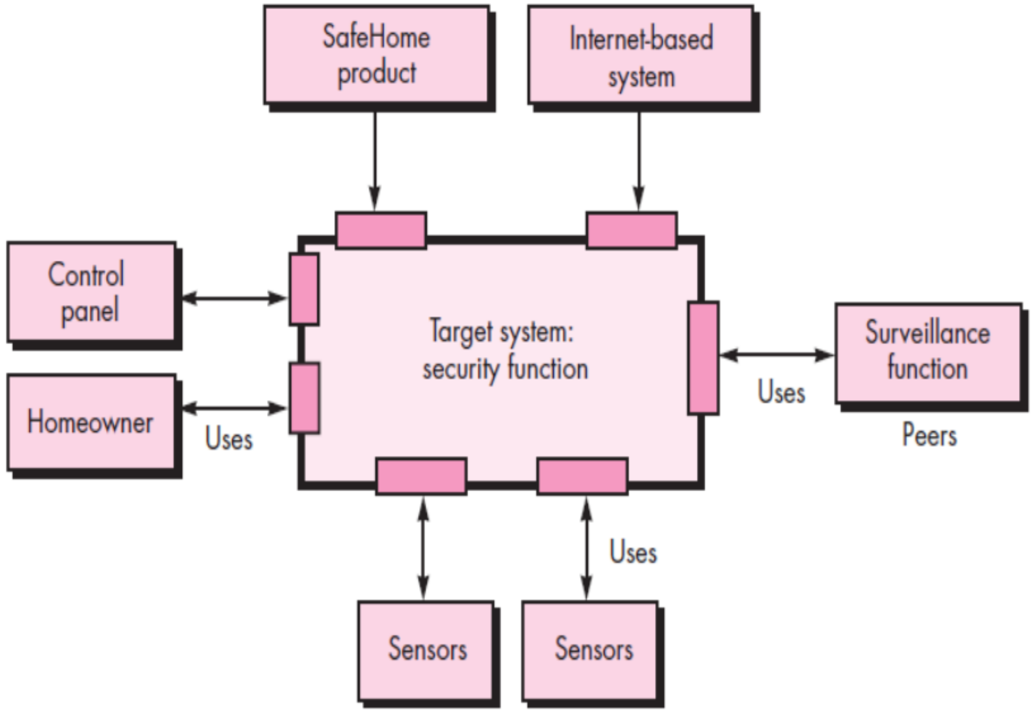
Architectural Context Diagram

An **Architectural Context Diagram** is a high-level visual representation used in software architecture to show a system’s **boundaries** and its **interactions with external entities**. It answers the question: *“What does the system interact with, and at a very coarse level, how?”*

Architectural Context Diagram



Architectural Context Diagram



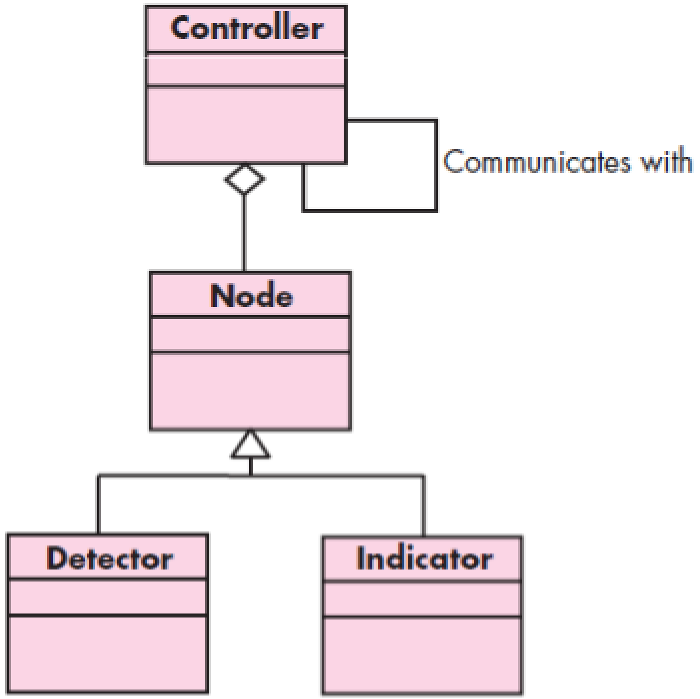
Defining the Archetypes

Defining the Archetypes

An **archetype** is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. Archetype is a recurring pattern or a standardized category of interaction that describes how external entities (users, systems, devices) interact with the system.

- Archetypes can be derived by examining the analysis classes defined as part of the requirements model.
- Think of the ACD as a map of who talks to the system. The Archetypes are the roles or categories those entities fall into.

Defining the Archetypes



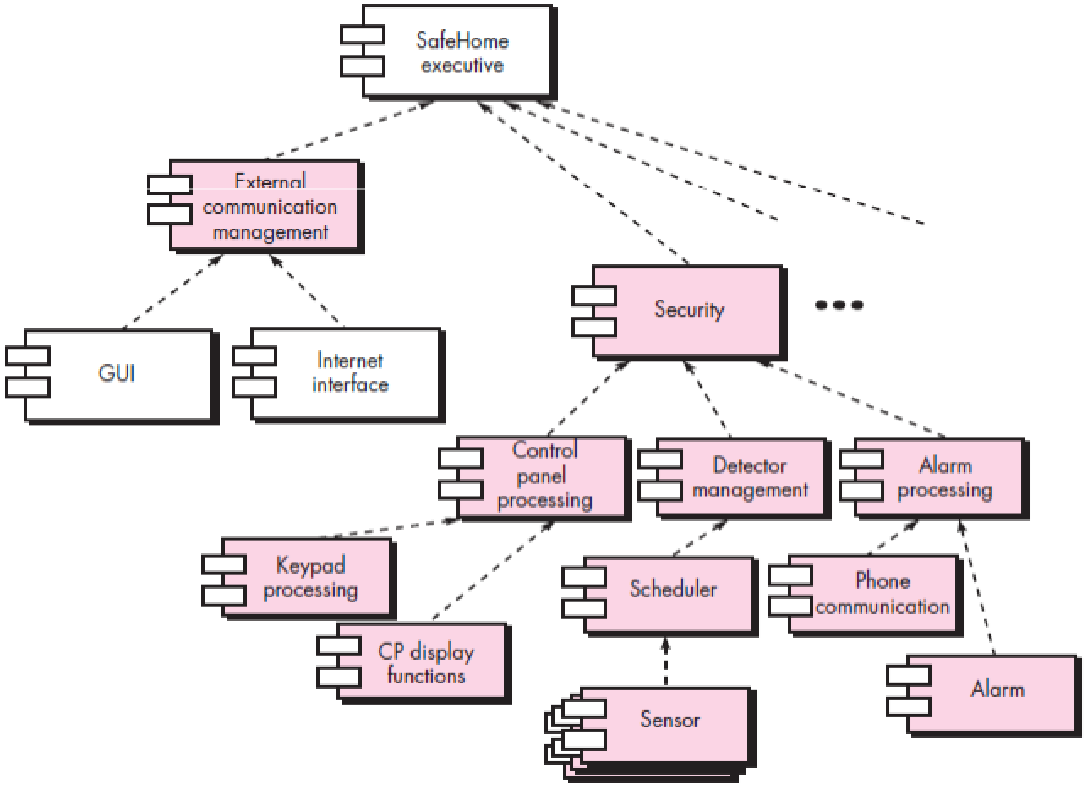
Refining the Architecture Into Components

Refining the Architecture into Components

Software architecture is refined into components. The components are derived from analysis class within application domain.

- Also, many infrastructure components are derived apart from application domain components. Eg. Memory management component.
- For eg, Based on the functionality, the following components are derived from SafeHome home security function: external communication management, control panel process, detector management, alarm processing.
- Design class would be defined after component level design.

Refining the Architecture into Components


SafeHome executive
External communication management
GUI
Internet interface
Security
•••
Control panel processing
Detector management
Alarm processing
Keypad processing
CP display functions
Scheduler
Phone communication
Sensor
Alarm