Unit-2

Architectural Styles and Patterns

Overview

2.1 Common architectural styles (Layered, Client-server, Microservices, Event-driven, SOA)

2.2 Architectural patterns (MVC, Broker, Pipe and Filter)

2.3 Design patterns (Singleton, Factory, Observer, Strategy)

2.4 Choosing appropriate architectural styles and patterns based on system requirements

2.5 Impact of architecture on non-functional requirements (performance, scalability, security)

Software Architecture

- A high-level structure of software system that includes the set of rules, patterns, and guidelines that dictate the organization, interactions and the component relationships.
- Serves as blueprint ensuring that the system meets it’s requirements and also maintainable, scalable and flexible.

Common Architectures

<u>1. Layered Architecture</u>

– Organizes system components into separate layers, where each layer has a specific role

– Commonly includes layers such as Presentation, Business Logic, Data Access, and Database

– Layers are stacked, and each layer only interacts with the layer directly below it

<u>Key Benefits:</u>

✓**Separation of Concerns:** Clear division between layers enhances modularity

✓**Maintainability:** Changes in one layer (e.g., data storage) don’t affect other layers

✓**Testability:** Easier to test in isolation due to modular structure

Common Architectures…

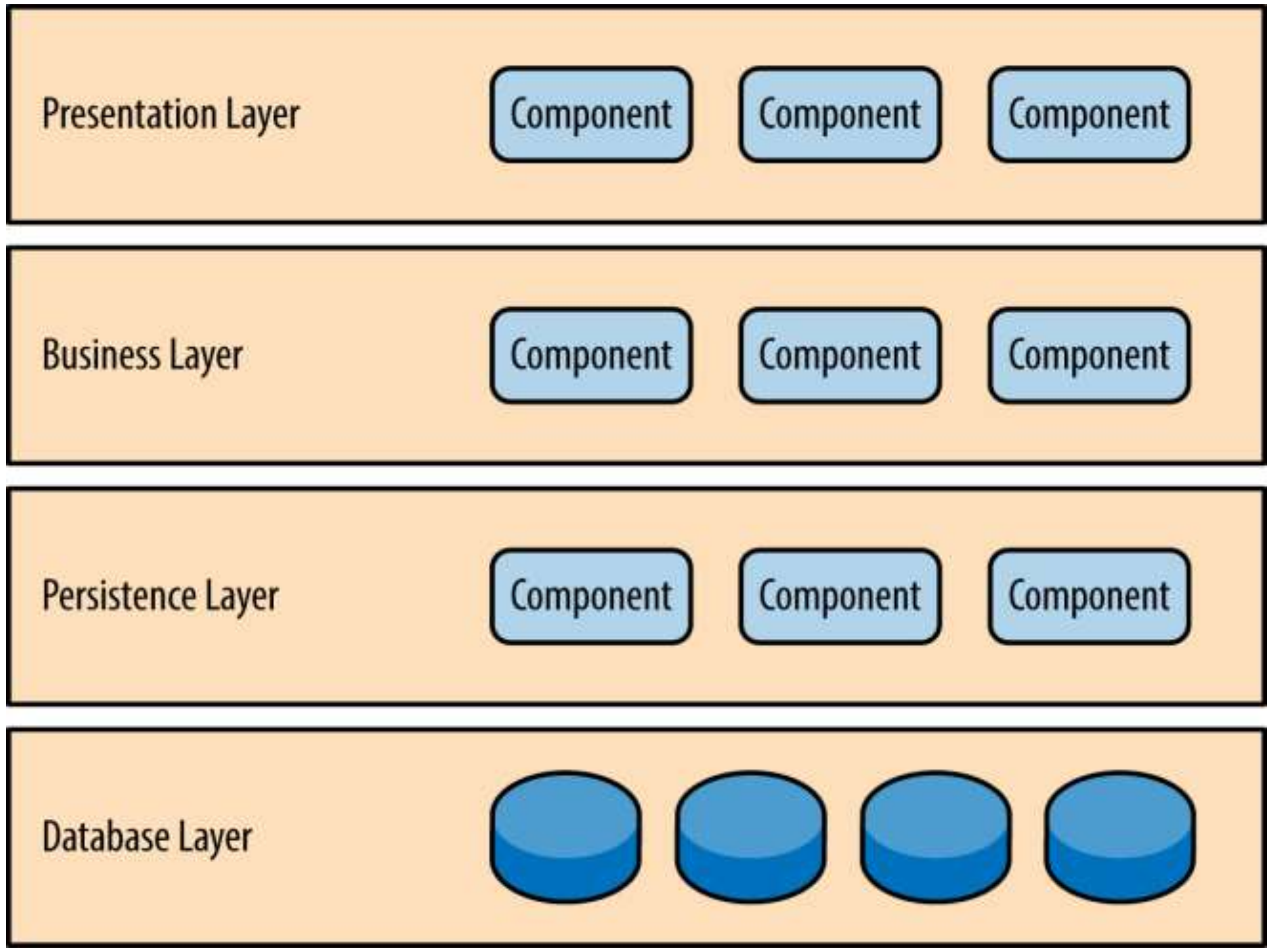
Drawbacks:

- **Rigidity:** Can be difficult to adapt or expand to meet scaling demands
- **Performance:** Extra layers can introduce latency

Common Use Cases:

Web applications, desktop applications with complex logic, and enterprise applications

Common Architectures…



User interface, when user interaction occurs

Core Application Logic

Manages data transactions and storage

Physical storage of data

Fig: Layered Architecture Style

Common Architectures…

2. Client-Server Architecture:

– A distributed model with clients (users or applications) making requests, and servers responding to them

– Clients and servers are separate; clients do not communicate directly with each other

Key Benefits:

- **Centralized Control:** Servers handle main data and resources, making it easier to manage
- **Scalability:** Supports multiple clients, as server can handle concurrent requests
- **Security:** Centralization makes it easier to implement security measures

Common Architectures…

<u>Drawbacks:</u>

- **Server Overload:** If demand is high, server can become a bottleneck
- **Single Point of Failure:** If the server goes down, clients lose access

<u>Common Use Cases:</u>

Web applications, file sharing systems, networked games, email servers

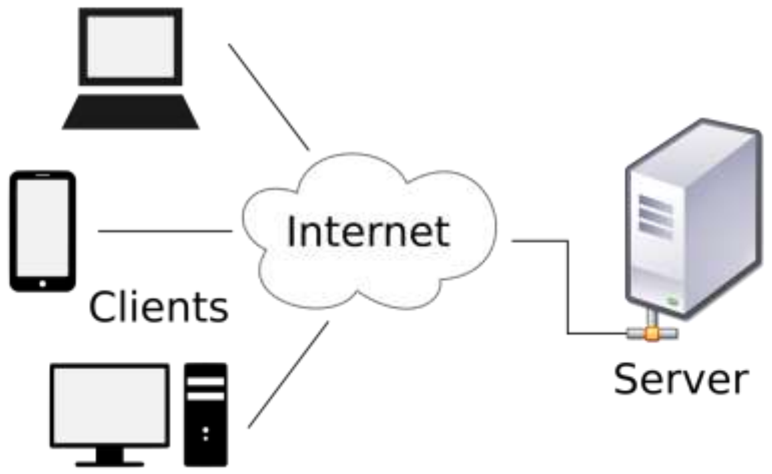


Fig: Client-Server Architecture

Common Architectures…

3. Microservices:

- An approach where the application is structured as a collection of loosely coupled, independently deployable services
- Each service handles a specific business function and communicates over a network (often via REST APIs)

Key Benefits:

- **Scalability:** Individual services can be scaled as needed
- **Fault Isolation:** Failure in one service doesn’t impact the entire system
- **Flexibility:** Allows for independent updates and deployments of services

Common Architectures…

Drawbacks:

- **Complexity:** More challenging to manage, monitor, and secure multiple services
- **Deployment Overhead:** Requires strong DevOps practices to handle continuous integration and deployment
- **Common Use Cases:**

  Large-scale applications, systems requiring rapid iteration, e-commerce platforms

Common Architectures…

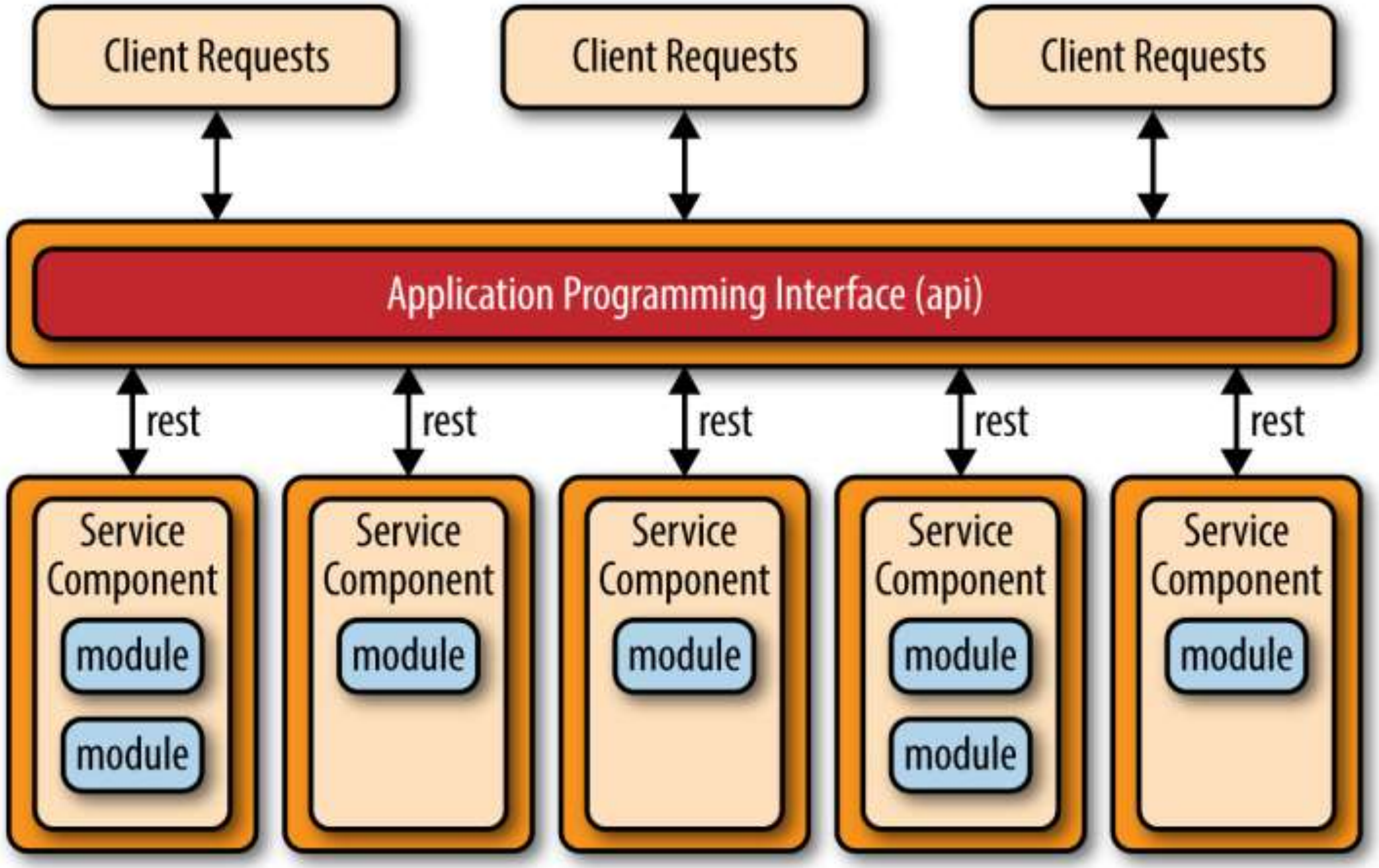


Fig: Microservices Architecture

Common Architectures…

4. Event-Driven Architecture:

– Architecture where components respond to events or triggers, processing them in real-time or near-real-time

– Typically includes event producers, event processors, and event consumers

– Events may be processed through a central message broker, like Kafka or RabbitMQ

Key Benefits:

- **Scalability:** Easily scales with high loads and multiple event sources
- **Real-time Processing:** Ideal for applications needing instant data processing
- **Loose Coupling:** Components are highly decoupled, improving flexibility

Common Architectures…

Drawbacks:

- **Complexity in Error Handling:** Difficult to ensure consistency, especially with distributed components
- **Latency in Event Processing:** Real-time might lag if events queue up

Common Use Cases:

IoT applications, stock trading platforms, notification systems, e-commerce events

Common Architectures…

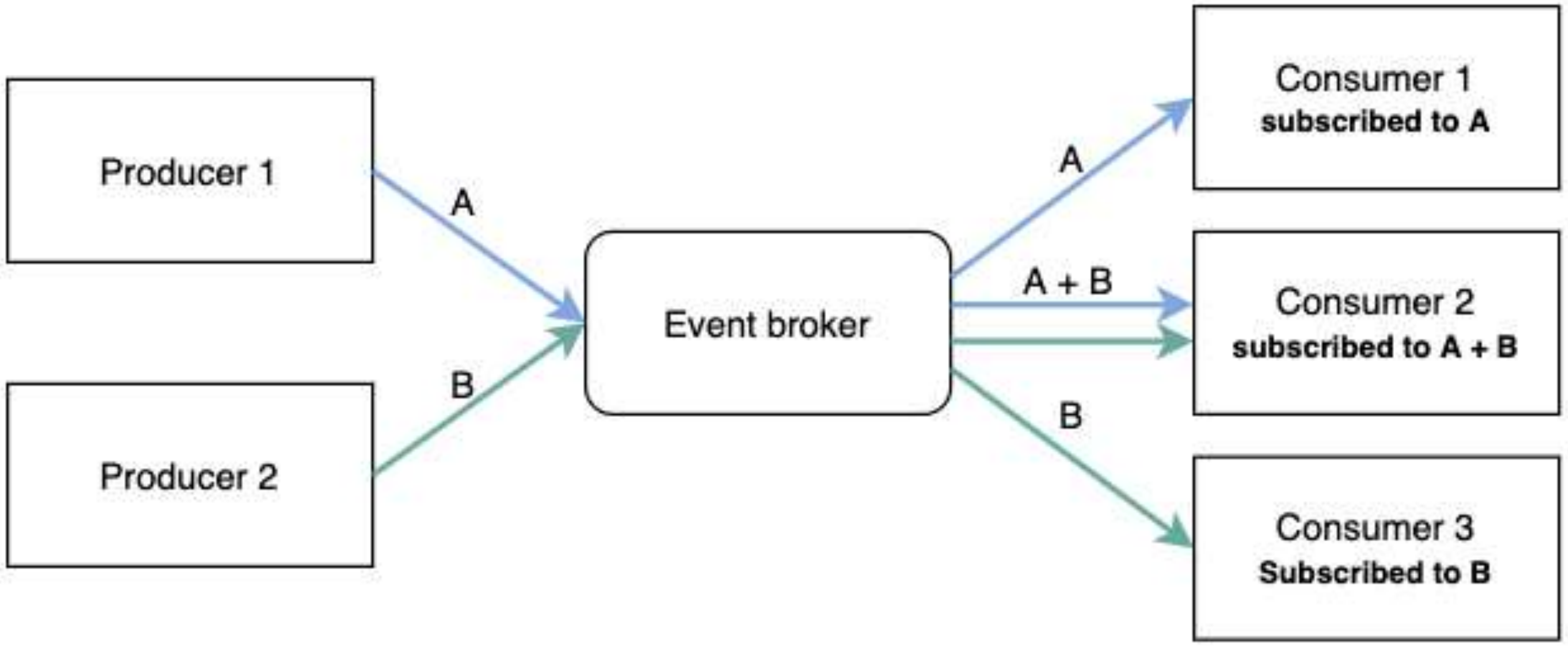


Fig: Event-Driven Architecture

Common Architectures…

5. Service-Oriented Architecture:

- A modular style that organizes functions as independent, reusable services, often communicating over a network
- Each service provides a business capability and is loosely coupled, typically with a centralized service bus for interaction

Key Benefits:

- **Reusability:** Services can be reused across different applications
- **Flexibility:** Easy to adapt, scale, or replace services
- **Scalability:** Can scale services based on need

Common Architectures…

Drawbacks:

- **Complex Governance:** Requires careful design and management of services and communication
- **Latency:** Network-based communication can introduce delays

Common Use Cases:

Large enterprises, applications with complex business requirements, and systems needing interoperability (e.g., integrating legacy and new applications)

Common Architectures…

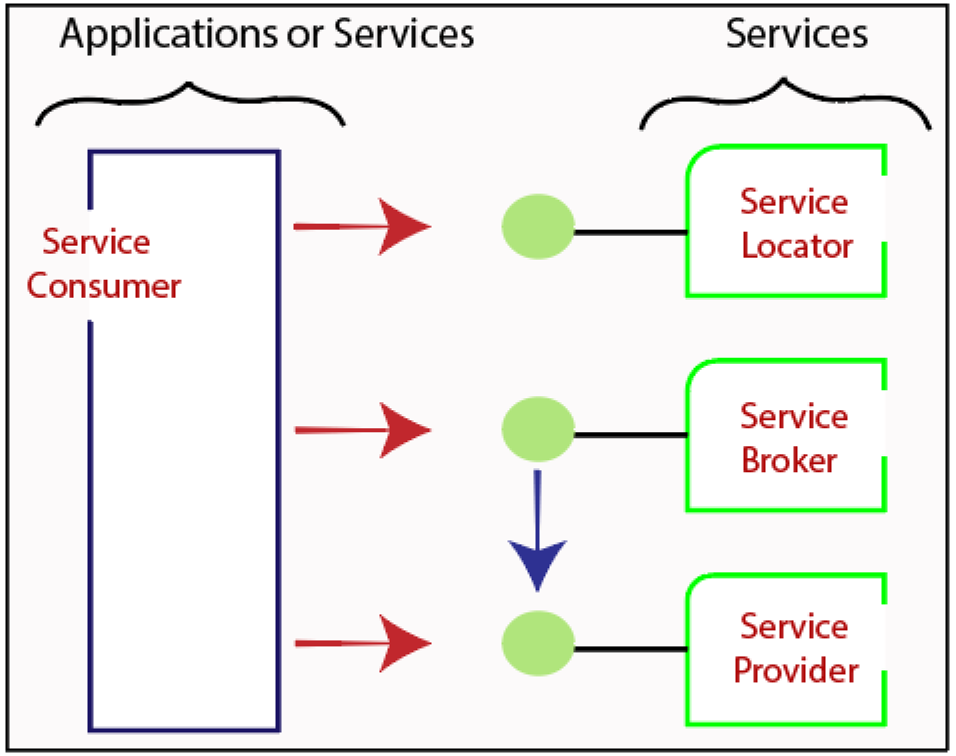


Fig: Service-Oriented-Architecture

Architectural patterns

– Reusable Solutions to recurring design problems within a specific architectural style.

– Gives the organization of high level system components.

Architectural patterns…

1. MVC =>(UI Intensive)

– Modular architecture for interactive systems.

– Promotes separation of concerns, improving testability and parallel development.

Components:

- Model: Encapsulates core data and logic
- View: Presents data to the user
- Controller: Manages input and coordinates between model and view.

Uses:

- Application with user interface (e.g. desktop, mobile, web)
- Scenarios where updates to UI and business logic occurs independently.

Architectural patterns…

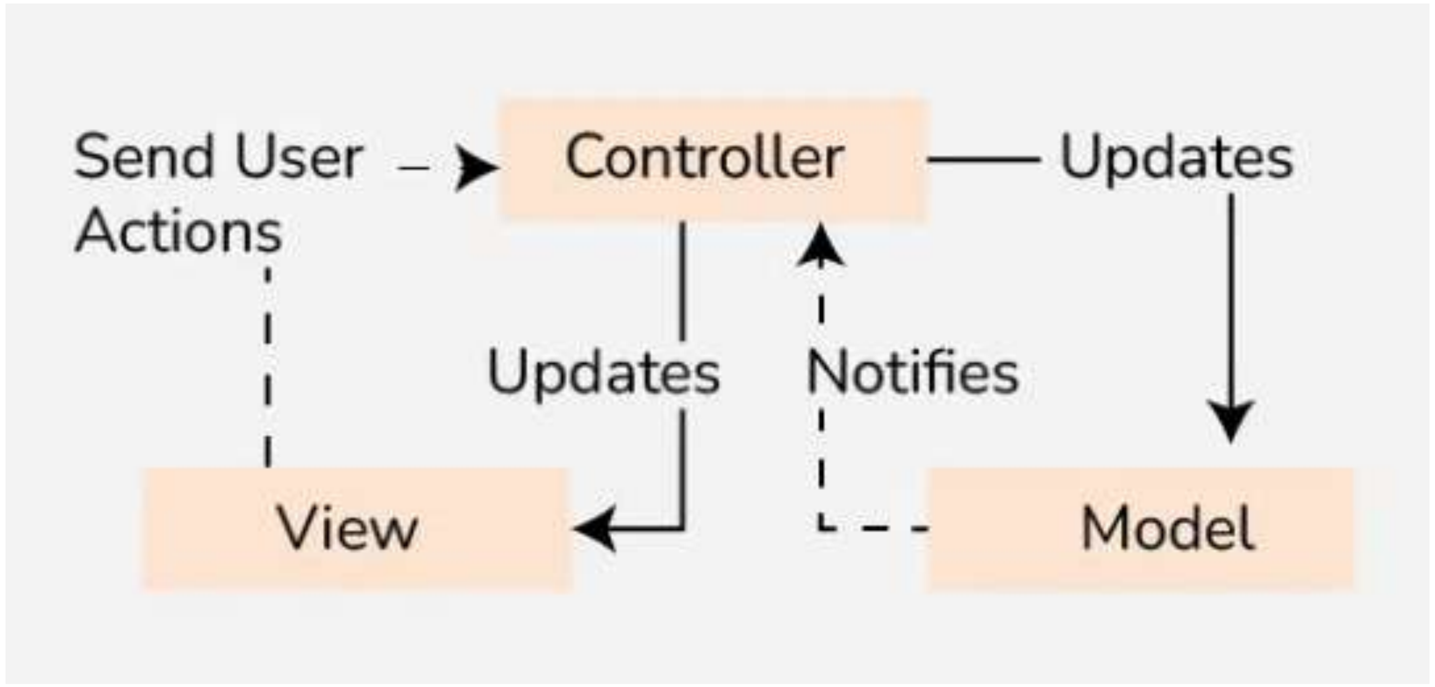


Fig: MVC Architecture

Architectural patterns…

Example: Django (Python), Laravel (PHP), ASP.NET(C#)

Strengths:

- Modular design simplifies debugging and testing.
- Encourages separation of responsibilities.
- Flexible for UI changes.

Weaknesses:

- Complexity in coordination between components.
- Overhead in small projects.

Architectural patterns…

2. Broker Pattern

- Suitable for systems where services are distributed across different locations.

Components:

- **Clients**: Request services.
- **Broker**: Mediates requests and locates service providers.
- **Service Providers**: Execute requests and return results.

Uses:

- Middleware-based systems.
- Distributed applications where components are loosely coupled.

Architectural patterns…

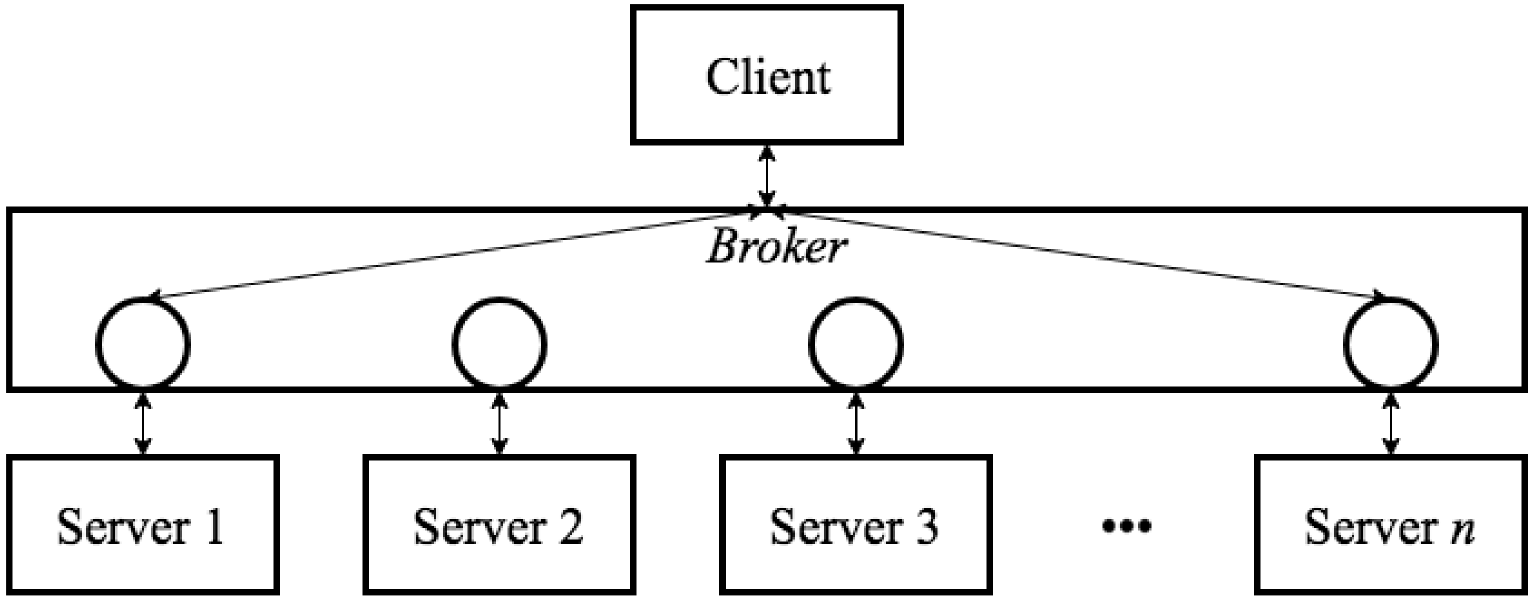


Fig: Broker pattern

Architectural patterns…

Example: Remote Procedure Calls (RPC), CORBA, and gRPC.

Strengths:

- Supports distributed system scalability.
- Decouples client and service logic.
- Centralized management improves service discovery.

Weaknesses:

- Broker failure can affect the entire system.
- Latency due to indirection.

Architectural patterns…

3. Pipe and Filter Pattern

- Suitable for systems processing data streams in a series of transformations.

Key Components:

Filters: Process and transform data.

Pipes: Connect filters and pass data between them.

Uses: Systems requiring modular processing, e.g., data analytics, compilers.

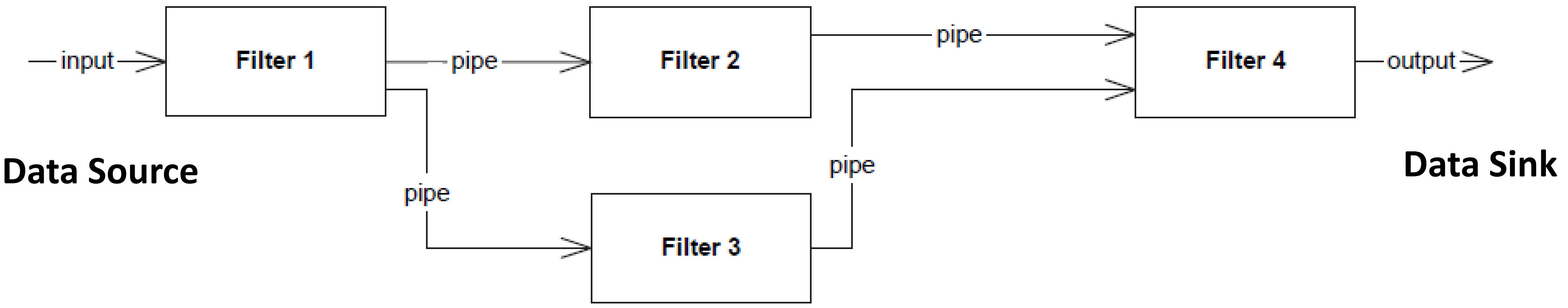


Fig: Pipe and Filter Architecture

Architectural patterns…

Example: Linux Command pipelines (cat|grep|sort)

Strengths:

- High modularity and reusability.
- Easy to scale and parallelize.
- Simplifies debugging by isolating issues in individual filters.

Weaknesses:

- Overhead in data transfer between filters.
- Latency in long pipelines.

Design patterns

Software design patterns are important tools for developers, providing proven solutions to common problem encountering during software development.

Benefits:

- Improves code maintainability
- Enhances scalability and flexibility
- Promotes best practices

Types

– Creational Design Pattern

– Structural Design Pattern

– Behavioral Design Pattern

Design patterns…

1. Creational Design Pattern

- *focus on the process of object creation or problems related to object creation.*
- *They help in making a system independent of how its objects are created, composed and represented.*

Types

- Singleton Pattern
- Factory Method Pattern
- Abstract Factory Method Pattern
- Builder Pattern
- Prototype Pattern

Design patterns…

2. Structural Design pattern

- *solves problems related to how classes and objects are composed/assembled to form larger structures which are efficient and flexible in nature.*
- *Structural class patterns use inheritance to compose interfaces or implementations.*

Types:

Adapter Pattern

Bridge Pattern

Composite Pattern

Decorator Pattern

Façade pattern

Proxy Pattern

Flyweight Pattern

Design patterns…

3. Behavioral Design Pattern

- *concerned with algorithms and the assignment of responsibilities between objects.*
- *Describe not just patterns of objects or classes but also the patterns of communication between them.*
- *These patterns characterize complex control flow that's difficult to follow at run-time.*

Types:

- *Observer Pattern*
- *Strategy Pattern*
- *State Pattern*
- *Command Pattern*
- *Template Pattern*
- *Interpreter Pattern*
- *Visitor Pattern*
- *Mediator Pattern*
- *Memento Pattern*

Singleton Pattern

- Ensures a class has only one instance and provides a global access point to it.

Intent:

- Prevent multiple instances of a class.
- Save memory and avoid inconsistencies.

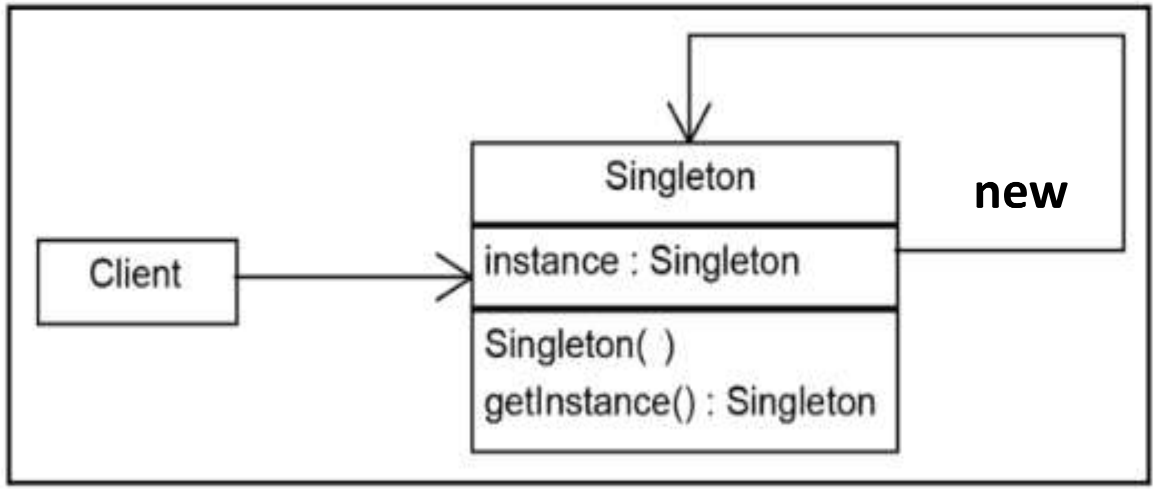


Fig: Singleton Pattern

Singleton Pattern…

Key Features:

- Private constructor.
- Static instance variable.
- Thread-safety considerations (e.g., double-checked locking).

Use Case:

- Logging services.
- Configuration settings.

Singleton Pattern…

Example:

```
public class Singleton {
    private static Singleton instance;

    private Singleton() { }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Factory Pattern

- Creates objects without exposing the information logic to the client, allowing more flexibility.
- Provide an interface for creating objects without specifying their concrete classes.

Intent:

- Delegates instantiation logic.
- Promotes loose coupling.

Factory Pattern…

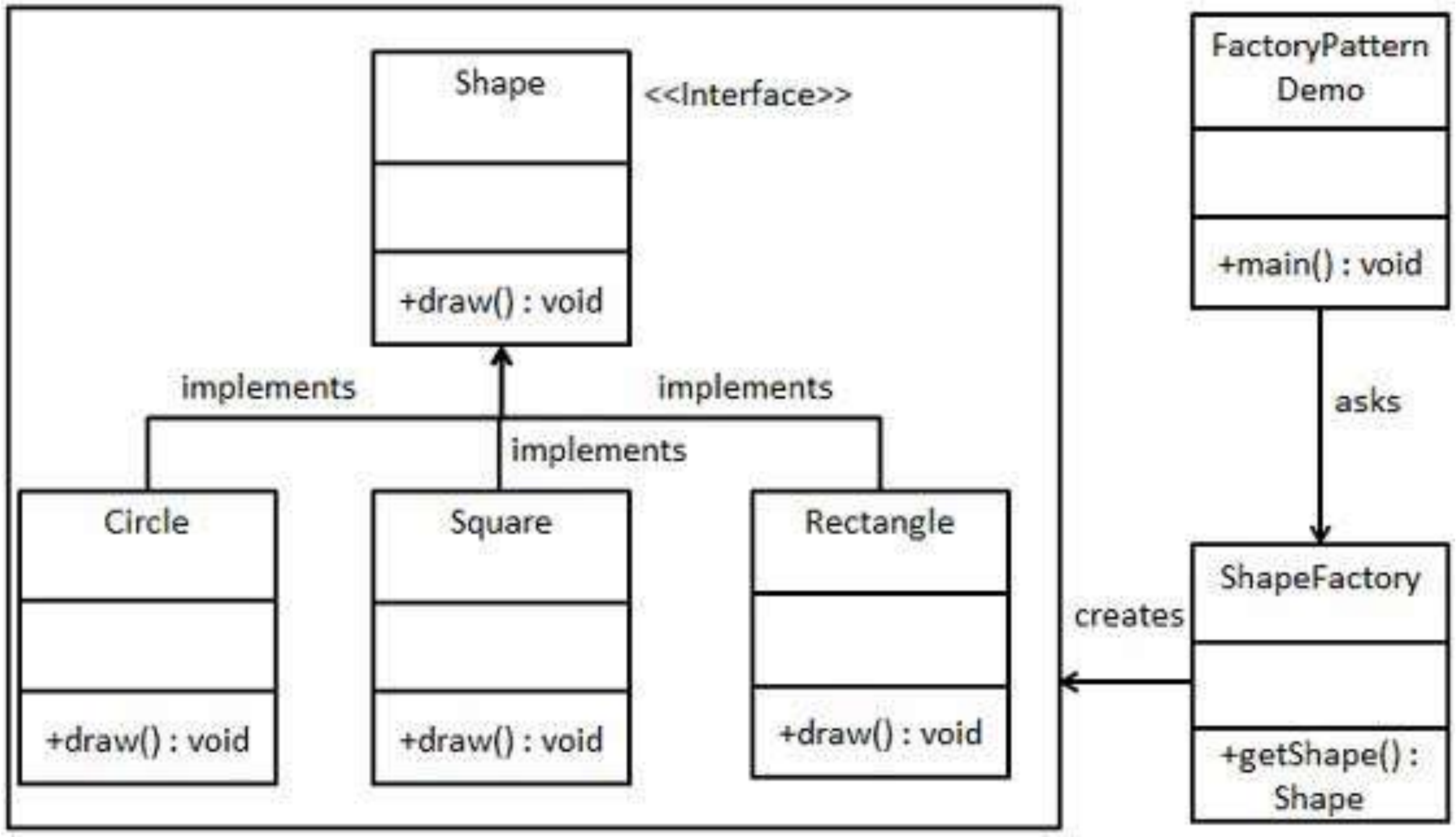


Fig: Factory Pattern

Factory Pattern…

```
public interface Shape {
   void draw();
}

public class Circle implements Shape {
   public void draw() {
      System.out.println("Drawing a Circle");
   }
}

public class Rectangle implements Shape {
   public void draw() {
      System.out.println("Drawing a Rectangle");
   }
}
```

```
public class ShapeFactory {
   public static Shape getShape(String type) {
      if (type.equals("Circle")) return new Circle();
      if (type.equals("Rectangle")) return new Rectangle();
      return null;
   }
}
```

Factory Pattern…

- **Key Features:**
  - Abstracts object creation.
  - Reduces code duplication.
- **Use Case:**
  - GUI frameworks (e.g., buttons, text fields).
  - Database connections.

Observer Pattern

- Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.
- **Intent:**
  - Decouple subject and observers.
  - Enable dynamic subscription.

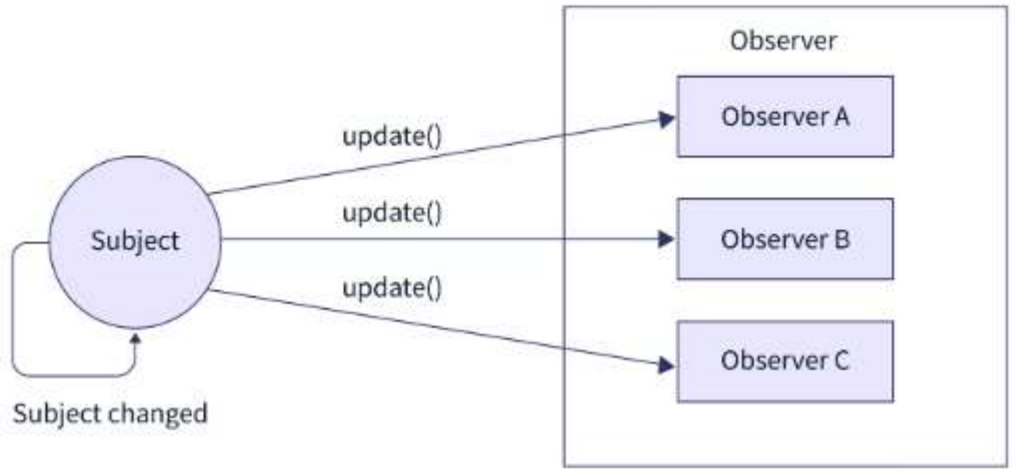


Fig: Observer Design Pattern

Observer Pattern…

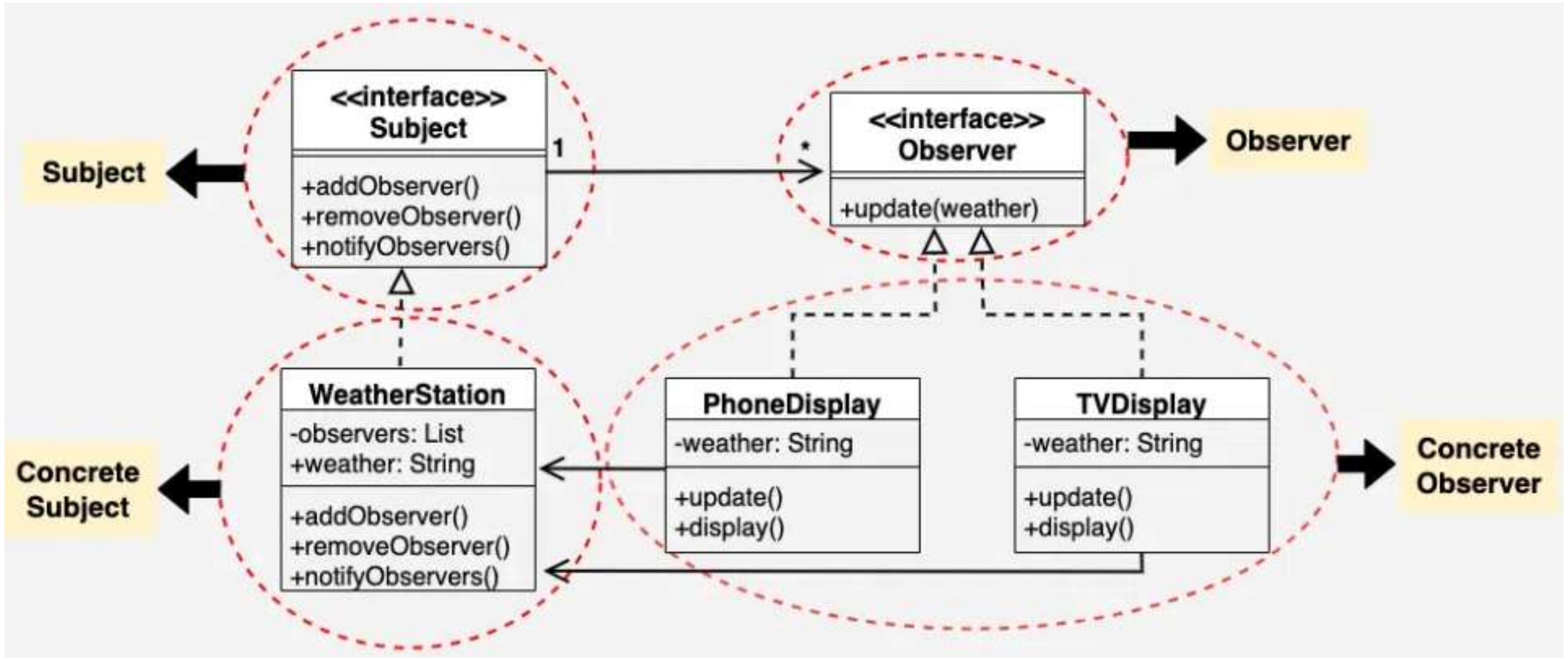


Fig: Observer Design Pattern

Observer Pattern…

subject interface

```
interface Subject {
   void addObserver(Observer o);
   void removeObserver(Observer o);
   void notifyObservers();
}
```

observer interface

```
interface Observer {
   void update(String weather);
}
```

Observer Pattern…

<u>WeatherStation Class (Concrete Subject)</u>

```
import java.util.ArrayList;
import java.util.List;

class WeatherStation implements Subject {
    private List<Observer> observers; // List to hold observers
    private String weather;           // Current weather information

    public WeatherStation() {
        observers = new ArrayList<>();
    }

    @Override
    public void addObserver(Observer o) {
        observers.add(o);
    }

    @Override
    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    @Override
    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(weather);
        }
    }

    // Method to update weather and notify observers
    public void setWeather(String weather) {
        this.weather = weather;
        notifyObservers();
    }
}
```

Observer Pattern…

PhoneDisplay Class (Concrete Observer)

```
class PhoneDisplay implements Observer {
   private String weather; // Stores the current weather

   @Override
   public void update(String weather) {
      this.weather = weather;
      display();
   }

   public void display() {
      System.out.println("Phone Display: Current weather is " + weather);
   }
}
```

Observer Pattern…

TVDisplay Class (Concrete Observer)

```
class TVDisplay implements Observer {
    private String weather; // Stores the current weather

    @Override
    public void update(String weather) {
        this.weather = weather;
        display();
    }

    public void display() {
        System.out.println("TV Display: Current weather is " + weather);
    }
}
```

Observer Pattern…

Testing Observer pattern

```
public class Main {
    public static void main(String[] args) {
        // Create the subject
        WeatherStation weatherStation = new
WeatherStation();

        // Create observers
        PhoneDisplay phoneDisplay = new
PhoneDisplay();
        TVDisplay tvDisplay = new TVDisplay();

        // Add observers to the subject
        weatherStation.addObserver(phoneDisplay);
        weatherStation.addObserver(tvDisplay);

        // Simulate weather updates
        weatherStation.setWeather("Sunny");
        weatherStation.setWeather("Rainy");
    }
}
```

Observer Pattern…

- **Key Features:**
  - Subject and Observer interfaces.
  - Push or pull model.
- **Use Case:**
  - Event-driven systems (e.g., GUI listeners).
  - Messaging services.

Strategy Pattern

- Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- Enables selecting an algorithm’s behavior at runtime.
- Different algorithms can be swapped in and out without altering the code that uses them.

Intent:

- Eliminate conditional logic.
- Promote open/closed principle.

Strategy Pattern…

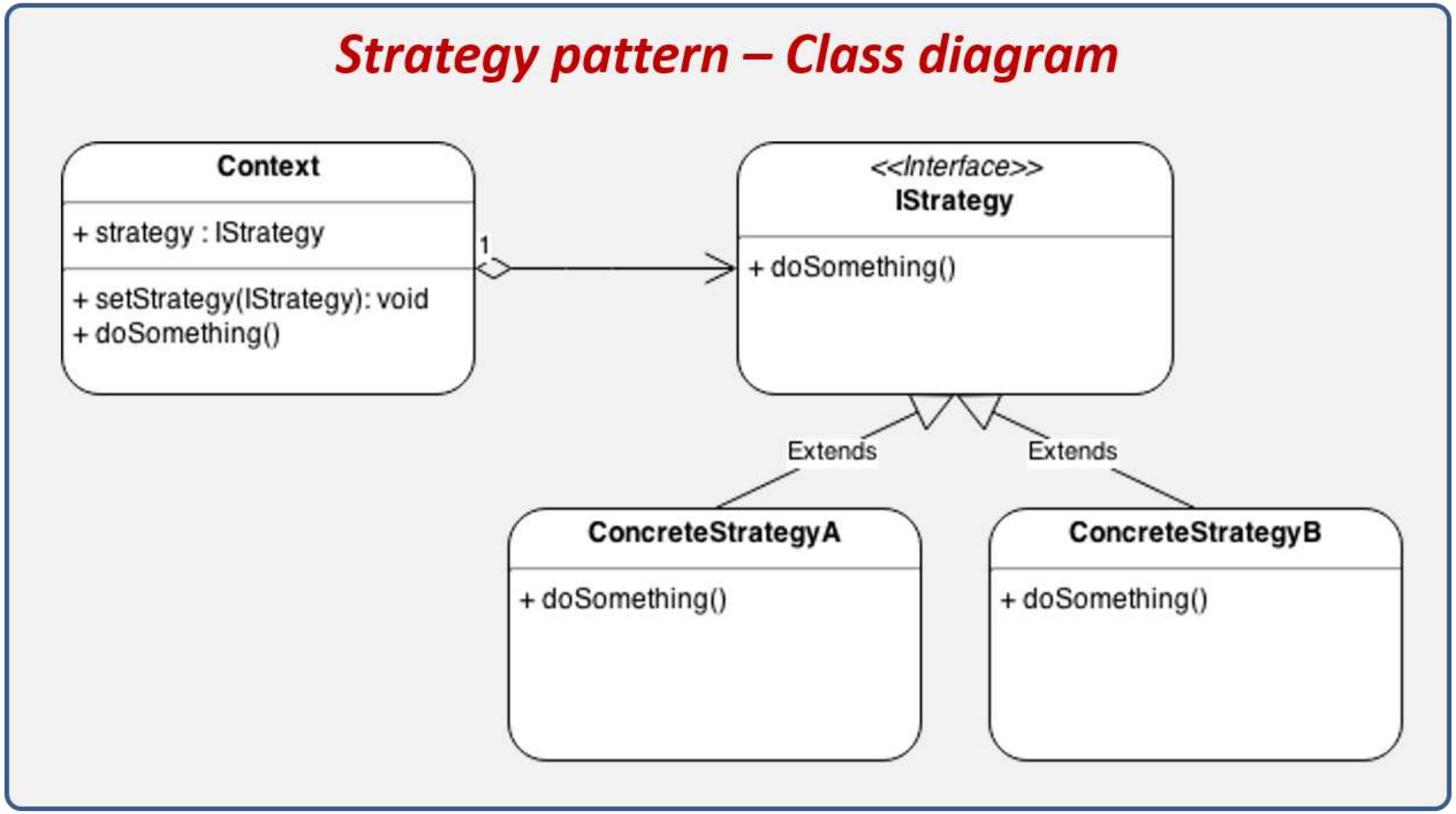


Fig: Strategy Pattern

Strategy Pattern…

```
public interface Strategy {
    int execute(int a, int b);
}

public class AddStrategy implements Strategy {
    public int execute(int a, int b) {
        return a + b;
    }
}

public class MultiplyStrategy implements Strategy {
    public int execute(int a, int b) {
        return a * b;
    }
}
```

```
public class Context {
    private Strategy strategy;

    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public int executeStrategy(int a, int b) {
        return strategy.execute(a, b);
    }
}
```

Strategy Pattern…

- **Key Features:**
  - Encapsulation of algorithms.
  - Dependency injection.
- **Use Case:**
  - Payment processing systems.
  - Sorting algorithms.

Discussions…

Q 1. How do design patterns differ from algorithms?

Design patterns focus on solving recurring design problems at the architectural or structural level, emphasizing the organization of code. Algorithms, on the other hand, are step-by-step procedures for solving specific problems at the computational level.

Q 2. Are design patterns language-specific?

No, design patterns are not tied to a specific programming language. They are conceptual solutions that can be implemented in various languages. However, the syntax and implementation details may vary.

Q 3. How do design patterns differ from architectural patterns?

Design patterns address specific design issues at a lower level, focusing on object creation, composition, and interaction. Architectural patterns, on the other hand, deal with higher-level structures of an entire application or system.

Choosing appropriate architectural styles and pattern based on system requirements

Choosing Architectural Styles and Pattern…

How?

1. Understand Functional and Non-Functional requirement
2. Identify constraints(budget, time, technology stack)
3. Map requirements to appropriate patterns(architectural & design)

Choosing Architectural Styles and Pattern…

Example 1 - E-commerce Platform

- **Functional:**
  - User authentication, product browsing, and checkout.
  - Real-time inventory updates.
- **Non-functional:**
  - Scalability to handle high traffic.
  - High availability and minimal downtime.

Choosing Architectural Styles and Pattern…

Example 2 - Real-Time Messaging App

- **Functional:**
  - Instant messaging with typing indicators.
  - User presence status and file sharing.
- **Non-functional:**
  - Performance: Low-latency message delivery.
  - Scalability: Handle millions of concurrent users.

Choosing Architectural Styles and Pattern…

Example 3 - Online Banking System

- **Functional:**
  - User authentication and account management.
  - Real-time fund transfers and transaction history.
- **Non-functional:**
  - Security: Protect sensitive user data.
  - Reliability: Ensure transactions are never lost.

Application Type	Architectural Style
Shared Memory	1. Blackboard 2. Data-centric 3. Rule-based
Distributed System	1. Client-server 2. Space based architecture 3. Peer-to-peer 4. Shared nothing architecture 5. Broker 6. Representational state transfer 7. Service-oriented
Messaging	1. Event-driven 2. Asynchronous messaging 3. .Publish-subscribe
Structure	1. Component-based 2. Pipes and filters 3. Monolithic application based 4. Layered
Adaptable System	1. Plug-ins 2. Reflection 3. Microkernel
Modern System	1. Architecture for Grid Computing 2. Multi-tenancy Architecture 3. Architecture for Big-Data

Impact of architecture on non-functional requirements

- **Performance** – will architecture handle expected loads?
- **Scalability** – can it grow with the user base?
- **Security** – strengthened by isolation
- **Maintainability** – how easy is it to update and modify?
- **Deployment** – what are the deployment complexities?

<u>Proper Architectural Choice</u>

↓

Ensure a system needs both current and future needs

Architecture styles evaluation

- The evaluation results are displayed in Table Next Slide. The method of measuring architecture styles utilized by Galster et al., (2010) has been replicated.
  - **“++”** represent a architecture style perform very well with some specific Quality Attribute;
  - **“+”** stands for some support;
  - **“–”** indicates that the style has negative impact on some specific Quality Attributes;
  - **“--”** indicates that very negative impact on a Quality Attributes;
  - **“o”** means no support, neutral or unsure.

STYLES \ QAs		Maintainability	Testability	Portability	Flexibility	Reusability	Simplicity	Availability	Security	Performance	Concurrency	Reliability	Scalability	Cost	Life Time	Usability	Supportability
Data Flow System	Batch Sequential	o	++	o	o	+	+	o	o	o	—	o	o	o	o	- -	o
	Pipe & Filter	+	+	o	+	+	+	o	o	o	++	o	+	o	o	- -	—
	Process control	o	o	+	o	o	o	o	o	o	o	o	o	+	o	o	o
Centralized Data Store System	Repository Arch	–	o	o	—	+	o	+	o	o	o	—	+	—	o	—	o
	Blackboard Arch	–	—	—	o	+	o	o	o	+	++	o	+	o	o	o	o
Large/ Complex System	Repository Arch	–	o	o	—	o	o	+	o	o	o	—	+	—	o	-	o
	Blackboard Arch	–	—	—	o	+	o	o	o	+	++	o	+	o	o	o	o
	Main-subroutine	–	o	o	o	—	o	o	-	o	o	+	o	o	o	o	o
	Master-slave	o	o	o	o	o	o	o	o	o	+	++	o	o	o	o	o
	Layered Arch	++	+	++	+	+	—	+	o	- -	—	o	+	o	o	o	o
Web Service	Service-Oriented	+	o	o	o	++	—	+	o	o	o	o	+	+	o	o	o
	MVC	+	o	o	+	o	+	+	o	o	o	o	- -	o	o	+	+
Distributed System	Client Server	—	—	o	o	+	+	—	++	—	o	—	+	o	o	o	o
	Broker Arch	++	—	+	+	+	+	o	—	—	o	o	o	o	o	o	o
	Service Oriented	+	o	o	o	++	—	+	o	o	o	o	+	+	o	o	o
User-Interaction Oriented System	MVC	+	o	o	+	o	+	+	o	o	o	o	- -	o	o	+	+
	Presentation Abstraction Control (PAC)	+	o	+	+	+	—	+	o	—	+	o	o	o	o	+	o

Table 4.2 Commonly used architecture styles in category and evaluation (Qian et al., 2008; Microsoft Application Architecture Guide, 2009 ; Qin et al., 2007; Zhu, 2005)