Behavioral design patterns are a category of design patterns that focus on how objects interact and communicate with each other, managing the flow of control and assignment of responsibilities in a flexible and efficient way. These patterns help define clear roles and algorithms governing communication between objects, promoting loose coupling and easier maintenance in software systems.

**Key Concepts of Behavioral Patterns:**

* They deal with object collaboration and communication.
* They define how requests, commands, and notifications are passed between objects.
* They help in reducing complex control flows among objects.
* They promote flexibility in behavior change without altering the objects themselve

**Observer Pattern**

**What is it?**

The Observer pattern defines a one-to-many dependency between objects so that when one object (called the subject) changes state, all its dependent objects (called observers) are notified and updated automatically.

**Key points:**

* The subject maintains a list of observers.
* Observers register themselves to be notified of changes.
* When the subject changes, it notifies all registered observers.
* It promotes loose coupling; the subject doesn't need to know details about the observers.

**Real-world analogy:**

A weather station (subject) notifies multiple smart devices (observers) when weather data changes.

import java.util.ArrayList;

import java.util.List;

// Subject interface

interface Subject {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

// Observer interface

interface Observer {

void update(float temperature);

}

// Concrete Subject

class WeatherStation implements Subject {

private List<Observer> observers = new ArrayList<>();

private float temperature;

public void registerObserver(Observer o) {

observers.add(o);

}

public void removeObserver(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(temperature);

}

}

public void setTemperature(float temperature) {

this.temperature = temperature;

notifyObservers();

}

}

// Concrete Observer

class PhoneDisplay implements Observer {

private float temperature;

public void update(float temperature) {

this.temperature = temperature;

System.out.println("Phone Display: Temperature updated to " + temperature);

}

}

// Usage

public class ObserverPatternDemo {

public static void main(String[] args) {

WeatherStation station = new WeatherStation();

PhoneDisplay phoneDisplay = new PhoneDisplay();

station.registerObserver(phoneDisplay);

station.setTemperature(25.5f);

station.setTemperature(26.0f);

}

}

**2. Strategy Pattern**

**What is it?**

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. The client can dynamically change the algorithm (or behavior) it uses at runtime without modifying the code.

Defiens a family of algorithms put each of them in a separate class and makes their objects interchangeable

**Real-world analogy:**

A navigation app can switch between different routing strategies: fastest route, shortest route, or avoid tolls.

// Strategy interface

interface PaymentStrategy {

void pay(int amount);

}

// Concrete Strategies

class CreditCardPayment implements PaymentStrategy {

public void pay(int amount) {

System.out.println("Paid " + amount + " using Credit Card.");

}

}

class PayPalPayment implements PaymentStrategy {

public void pay(int amount) {

System.out.println("Paid " + amount + " using PayPal.");

}

}

// Context

class ShoppingCart {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void checkout(int amount) {

paymentStrategy.pay(amount);

}

}

// Usage

public class StrategyPatternDemo {

public static void main(String[] args) {

ShoppingCart cart = new ShoppingCart();

// Use credit card payment

cart.setPaymentStrategy(new CreditCardPayment());

cart.checkout(100);

// Switch to PayPal payment

cart.setPaymentStrategy(new PayPalPayment());

cart.checkout(200);

}

}

Command Pattern

eal-Life Example

Think of a TV remote:

* The Remote is the Invoker — it doesn’t know *how* the TV works.
* Each button press is a Command — “turn on the TV” or “turn off the TV”.
* The TV is the Receiver — it knows how to actually turn on/off.

Invoker → executes Command object → Command object calls Receiver’s method