Design Patterns in Java

Design patterns are typical solutions to common problems in software design. They represent best practices used by experienced object-oriented software developers. They can speed up the development process by providing tested, proven development paradigms.

Key Concepts

1. **Reusability**: Design patterns promote code reuse.
2. **Flexibility**: They help make a system independent of how its objects are created, composed, and represented.
3. **Maintainability**: Patterns improve the readability and maintainability of the code base.
4. **Loosely Coupled Code**: Patterns encourage reducing dependencies between components.

Principles

1. **Encapsulation**: Hide the internal state and require all interaction to be performed through an object's methods.
2. **Single Responsibility Principle**: A class should have only one reason to change.
3. **Open/Closed Principle**: Software entities should be open for extension but closed for modification.
4. **Liskov Substitution Principle**: Subtypes must be substitutable for their base types.
5. **Interface Segregation Principle**: Many client-specific interfaces are better than one general-purpose interface.
6. **Dependency Inversion Principle**: Depend upon abstractions, not concretions.

Types of Design Patterns

Design patterns are classified into three main categories:

1. **Creational Patterns**: Deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.
2. **Structural Patterns**: Deal with object composition or the way to realize relationships between entities.
3. **Behavioral Patterns**: Deal with object collaboration and responsibility.

Creational	Structural	Behavioral
1. **Singleton Pattern** 2. **Factory Pattern** 3. **Abstract Factory Pattern** 4. **Builder Pattern** 5. **Prototype Pattern**	1. **Adapter Pattern** 2. **Composite Pattern** 3. **Proxy Pattern** 4. **Flyweight Pattern** 5. **Facade Pattern** 6. **Bridge Pattern** 7. **Decorator Pattern**	1. **Strategy Pattern** 2. **Observer Pattern** 3. **Chain of Responsibility Pattern** 4. **Command Pattern** 5. **Interpreter Pattern** 6. **Iterator Pattern** 7. **Mediator Pattern** 8. **Memento Pattern** 9. **State Pattern**

		10. **Template Method Pattern** 11. **Visitor Pattern**

Few Commonly Used Design Patterns

1. **Singleton Pattern**
    - **Description:** Ensures a class has only one instance and provides a global point of access to that instance.
    - **Use Case:** Database connections, configuration managers.
2. **Factory Method Pattern**
    - **Description:** Defines an interface for creating objects but allows subclasses to alter the type of objects that will be created.
    - **Use Case:** GUI libraries where different operating systems require different implementations.
3. **Abstract Factory Pattern**
    - **Description:** Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
    - **Use Case:** Creating different sets of user interfaces for different platforms.
4. **Builder Pattern**
    - **Description:** Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
    - **Use Case:** Building complex objects like HTML documents or complex configurations.
5. **Prototype Pattern**
    - **Description:** Creates new objects by copying an existing object, known as the prototype.
    - **Use Case:** Cloning objects in scenarios where the cost of creating new objects is higher.
6. **Adapter Pattern**

- **Description:** Allows incompatible interfaces to work together by providing a wrapper that converts the interface of a class into another interface expected by the client.
- **Use Case:** Integrating third-party libraries or legacy systems with modern applications.

7. **Observer Pattern**
   - **Description:** Defines a dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
   - **Use Case:** Implementing event handling systems, such as user interface event listeners.

9. **Decorator Pattern**
   - **Description:** Adds additional functionality to an object dynamically without altering its structure. It provides a flexible alternative to subclassing for extending functionality.
   - **Use Case:** Adding features to UI components, such as adding scrollbars or borders to windows.
10. **Chain of Responsibility Pattern**
    - **Description:** Passes a request along a chain of handlers. Each handler can either process the request or pass it to the next handler in the chain. It decouples sender and receiver of a request.
    - **Use Case:** Handling requests in a logging system, processing form submissions with validation rules.

Singleton Design Pattern

The Singleton Design Pattern ensures that a class has only one instance and provides a global point of access to that instance. This pattern is particularly useful when exactly one object is needed to coordinate actions across the system.

Key Characteristics:

1. **Single Instance:** Only one instance of the class is created.
2. **Global Access Point:** Provides a global point of access to the instance.
3. **Controlled Access:** Controls the instantiation process to ensure that no more than one instance is created.

How Does It Work?

1. **Private Constructor:** The class constructor is made private to prevent direct instantiation from outside the class.
2. **Static Instance:** A static instance of the class is maintained within the class itself.
3. **Static Method:** A public static method is provided to access the unique instance of the class.

Real-Time Use Cases:

1. **Configuration Management:**
   - **Example:** In many applications, configuration settings are often read from a file or database. Using a Singleton ensures that the configuration is read once and reused throughout the application, rather than loading it multiple times.
   - **Reason:** This approach avoids the overhead of multiple reads and ensures consistency in configuration data.
2. **Logging:**
   - **Example:** A logging system might need to be accessed from different parts of an application. A Singleton Logger ensures that all log messages are written through a single instance, preventing the creation of multiple log files or inconsistent log entries.
   - **Reason:** Centralized logging helps maintain consistency and manage resources effectively.
3. **Database Connections:**
   - **Example:** Managing database connections can be resource-intensive. A Singleton Database Manager ensures that only one connection instance is created and used throughout the application, which helps in managing connection pooling and avoiding excessive resource consumption.
   - **Reason:** Efficient connection management leads to improved performance and resource utilization.

Sample Program in Java:

Here's a simple example demonstrating the Singleton Design Pattern in Java:

```
// Singleton class
public class Singleton {
    // Private static variable to hold the single instance
    private static Singleton instance;

    // Private constructor to prevent instantiation
    private Singleton() {
        // Private constructor to restrict instantiation
    }

    // Public static method to provide access to the instance
    public static Singleton getInstance() {
        // Create instance if it does not exist
        if (instance == null) {
            instance = new Singleton();
        }
```

```
        return instance;
    }

    // Method to demonstrate Singleton functionality
    public void showMessage() {
        System.out.println("Hello from Singleton!");
    }
}

// Main class to test Singleton
public class Main {
    public static void main(String[] args) {
        // Get the only instance of Singleton
        Singleton singleton = Singleton.getInstance();

        // Call method on the Singleton instance
        singleton.showMessage();

        // Verify if the instances are the same
        Singleton anotherSingleton = Singleton.getInstance();
        System.out.println("Are both instances equal? " + (singleton ==
anotherSingleton));
    }
}
```

Explanation:

1. **Private Static Variable:** `private static Singleton instance;` holds the single instance of the class.
2. **Private Constructor:** `private Singleton()` prevents instantiation from outside the class.
3. **Public Static Method:** `public static Singleton getInstance()` provides access to the instance. It creates the instance only if it does not already exist.
4. **Instance Verification:** The output from `singleton == anotherSingleton` will be `true`, showing that both references point to the same instance.

Prototype Design Pattern

The Prototype Design Pattern is used to create new objects by copying an existing object, known as the prototype. This pattern is particularly useful when the cost of creating a new object is more expensive than copying an existing one. It helps in creating objects dynamically and efficiently.

Key Characteristics:

1. **Prototype:** Defines an interface for cloning itself.
2. **Cloning:** Allows for the creation of duplicate objects based on a prototype.
3. **Decoupling:** Client code is decoupled from the specific classes of objects it needs to create.

How Does It Work?

1. **Prototype Interface:** Defines a method for cloning itself.
2. **Concrete Prototype:** Implements the cloning method.
3. **Client Code:** Uses the prototype to create new instances by cloning rather than creating new objects from scratch.

Real-Time Use Cases:

1. **Object Pooling:**
   - **Example:** In a game development scenario, different types of characters or objects (e.g., enemies, bullets) might be frequently created and destroyed. Using prototypes allows for the efficient reuse of objects rather than creating and initializing new ones every time.
   - **Reason:** Reduces the overhead of object creation and improves performance by reusing existing objects.
2. **Configuration Management:**
   - **Example:** Applications often need to create multiple instances of configurations that are similar but with minor variations. Using prototypes allows creating new configuration objects based on a template.
   - **Reason:** Ensures consistency and reduces the complexity of creating new configurations from scratch.
3. **Document Editing:**
   - **Example:** In a document editor, you might need to create new documents that are similar to an existing template. The prototype pattern allows creating new documents based on a predefined template.
   - **Reason:** Facilitates creating new documents quickly by copying an existing document structure.

Sample Program in Java:

Here's a simple example demonstrating the Prototype Design Pattern in Java:

```
// Prototype interface
interface Prototype {
    Prototype clone();
}

// ConcretePrototype class
class ConcretePrototype implements Prototype {
    private String name;

    // Constructor
    public ConcretePrototype(String name) {
        this.name = name;
    }

    // Getter
    public String getName() {
        return name;
    }
```

```
    // Implement the clone method
    @Override
    public Prototype clone() {
        return new ConcretePrototype(this.name);
    }

    @Override
    public String toString() {
        return "ConcretePrototype{name='" + name + "'}";
    }
}

// Main class to test Prototype pattern
public class PrototypePatternDemo {
    public static void main(String[] args) {
        // Create an initial prototype object
        ConcretePrototype prototype1 = new ConcretePrototype("Prototype1");

        // Clone the prototype object
        ConcretePrototype clonedPrototype = (ConcretePrototype)
prototype1.clone();

        // Display the objects
        System.out.println("Original: " + prototype1);
        System.out.println("Cloned: " + clonedPrototype);

        // Verify if the objects are different instances but with the same
state
        System.out.println("Are both objects the same instance? " +
(prototype1 == clonedPrototype));
        System.out.println("Do both objects have the same name? " +
(prototype1.getName().equals(clonedPrototype.getName())));
    }
}
```

Explanation:

1. **Prototype Interface:** Defines a `clone()` method that returns a new instance of the object.
2. **ConcretePrototype Class:** Implements the `clone()` method, which creates a new object with the same state as the current object.
3. **Client Code:** Demonstrates creating a prototype object, cloning it, and verifying that the cloned object has the same state but is a different instance.

Factory Method Design Pattern

The Factory Design Pattern provides a way to create objects without specifying the exact class of the object that will be created. Instead of calling a constructor directly, clients use a

factory method to get an instance of the desired class. This pattern encapsulates the object creation logic, making it easier to manage and extend.

Key Characteristics:

1. **Factory Method:** A method or interface that defines the contract for creating objects.
2. **Product:** The objects created by the factory method.
3. **Concrete Products:** The specific implementations of the product that the factory method can create.

How Does It Work?

1. **Define a Product Interface:** An interface or abstract class that specifies the methods for the objects created by the factory.
2. **Concrete Products:** Classes that implement the product interface.
3. **Factory Interface:** Defines the factory method for creating products.
4. **Concrete Factory:** Implements the factory method to return instances of concrete products.

Real-Time Use Cases:

1. **GUI Frameworks:**
   - **Example:** In graphical user interface (GUI) frameworks, different operating systems might have different implementations of buttons, windows, etc. A factory method can be used to create GUI components appropriate for the current operating system.
   - **Reason:** This approach allows the same codebase to generate UI elements specific to different environments without changing the underlying logic.
2. **Database Connection Management:**
   - **Example:** A system might support different types of databases (e.g., MySQL, PostgreSQL, Oracle). A factory can be used to create database connection objects depending on the type of database being used.
   - **Reason:** Centralizes the creation logic for database connections, making it easier to switch between different databases.
3. **Document Generation:**
   - **Example:** In a document generation system, different types of documents (e.g., PDF, Word, Excel) might need to be created. A factory method can be used to create document objects based on the desired format.
   - **Reason:** Simplifies the creation process by delegating the responsibility to a factory, which manages the instantiation of different document types.

Sample Program in Java:

Here's a simple example demonstrating the Factory Design Pattern in Java:

```
// Vehicle interface
interface Vehicle {
    void drive();
}
```

```java
// Concrete Vehicle classes
class Car implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Driving a car");
    }
}

class Truck implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Driving a truck");
    }
}

// Factory interface
interface VehicleFactory {
    Vehicle createVehicle();
}

// Concrete Factory classes
class CarFactory implements VehicleFactory {
    @Override
    public Vehicle createVehicle() {
        return new Car();
    }
}

class TruckFactory implements VehicleFactory {
    @Override
    public Vehicle createVehicle() {
        return new Truck();
    }
}

// Main class to test Factory pattern
public class FactoryPatternDemo {
    public static void main(String[] args) {
        // Create a Car using CarFactory
        VehicleFactory carFactory = new CarFactory();
        Vehicle car = carFactory.createVehicle();
        car.drive();

        // Create a Truck using TruckFactory
        VehicleFactory truckFactory = new TruckFactory();
        Vehicle truck = truckFactory.createVehicle();
        truck.drive();
    }
}
```

Explanation:

1. **Vehicle Interface:** Defines a method `drive()` that all vehicle types will implement.

2. **Concrete Vehicle Classes:** `Car` and `Truck` implement the `Vehicle` interface and provide their specific implementations of the `drive()` method.
3. **Factory Interface:** `VehicleFactory` declares a method `createVehicle()` for creating `Vehicle` objects.
4. **Concrete Factories:** `CarFactory` and `TruckFactory` implement the `VehicleFactory` interface and return instances of `Car` and `Truck`, respectively.
5. **Client Code:** Demonstrates creating vehicles using the factories and invoking their methods.

Builder Design Pattern

The Builder Design Pattern is used when you need to construct an object that requires multiple steps or involves complex configurations. It helps in creating immutable objects with a clear and flexible way to construct different representations.

Key Characteristics:

1. **Builder:** Provides an interface for creating parts of a product.
2. **ConcreteBuilder:** Implements the Builder interface and constructs the product by assembling its parts.
3. **Director:** Constructs an object using the Builder interface. It directs the construction process.
4. **Product:** The complex object that is being built.

How Does It Work?

1. **Define the Product:** Create a class that represents the complex object to be built.
2. **Builder Interface:** Define an interface for building different parts of the product.
3. **Concrete Builder:** Implement the builder interface to build the parts of the product.
4. **Director:** Construct the product using a builder.
5. **Client Code:** Use the director to construct the product with the desired configuration.

Real-Time Use Cases:

1. **Building Complex Objects:**
    - **Example:** Constructing a complex document like a report or a web page with different sections (header, body, footer) and configurations.
    - **Reason:** The Builder Pattern helps manage the complexity of assembling different parts of the document.
2. **Creating Configuration Objects:**
    - **Example:** Creating a configuration object for a complex system, such as a network setup or a software deployment, where multiple settings need to be specified.
    - **Reason:** Allows flexible and controlled construction of configuration objects with various options.

3. **Product Construction with Optional Parameters:**
    - **Example:** Building a product with optional features like customizations for a vehicle (e.g., additional features, paint color, engine type).
    - **Reason:** Simplifies the process of creating products with different combinations of options without having to deal with a large number of constructors or setters.

Sample Program in Java:

Here's a simple example demonstrating the Builder Design Pattern in Java:

```
// Product class
class Computer {
    private String CPU;
    private String RAM;
    private String storage;
    private boolean isGraphicsCardEnabled;

    // Private constructor to enforce use of Builder
    private Computer(Builder builder) {
        this.CPU = builder.CPU;
        this.RAM = builder.RAM;
        this.storage = builder.storage;
        this.isGraphicsCardEnabled = builder.isGraphicsCardEnabled;
    }

    // Static nested Builder class
    public static class Builder {
        private String CPU;
        private String RAM;
        private String storage;
        private boolean isGraphicsCardEnabled;

        // Builder methods for setting optional values
        public Builder setCPU(String CPU) {
            this.CPU = CPU;
            return this;
        }

        public Builder setRAM(String RAM) {
            this.RAM = RAM;
            return this;
        }

        public Builder setStorage(String storage) {
            this.storage = storage;
            return this;
        }

        public Builder enableGraphicsCard(boolean isEnabled) {
            this.isGraphicsCardEnabled = isEnabled;
            return this;
        }

        // Build method to construct the Computer object
```

```
        public Computer build() {
            return new Computer(this);
        }
    }

    @Override
    public String toString() {
        return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", storage=" +
storage + ", GraphicsCardEnabled=" + isGraphicsCardEnabled + "]";
    }
}

// Main class to test Builder pattern
public class BuilderPatternDemo {
    public static void main(String[] args) {
        // Using the Builder to create a Computer object
        Computer computer = new Computer.Builder()
                .setCPU("Intel i9")
                .setRAM("16GB")
                .setStorage("1TB SSD")
                .enableGraphicsCard(true)
                .build();

        System.out.println(computer);
    }
}
```

Explanation:

1. **Product Class:** `Computer` represents the complex object being built. It has a private constructor to enforce the use of the Builder.
2. **Builder Class:** `Builder` is a static nested class within `Computer` that provides methods for setting different parts of the `Computer` object. It has a `build()` method that creates and returns a `Computer` instance.
3. **Client Code:** In the `BuilderPatternDemo` class, the `Builder` is used to construct a `Computer` object with specific configurations. The client code does not interact with the internal construction process but uses a fluent interface to specify the required attributes.

Adapter Design Pattern

The **Adapter Design Pattern** is a structural design pattern used to allow objects with incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces by providing a wrapper that translates calls from one interface into calls to another.

Key Components:

1. **Target:** The interface that the client expects to work with.
2. **Adapter:** The class that implements the Target interface and translates requests to the Adaptee.

3. **Adaptee:** The class that has the functionality but uses an incompatible interface.
4. **Client:** The class that interacts with the Target interface.

How It Works:

1. **Target Interface:** Defines the interface that the client expects.
2. **Adaptee:** Has an existing interface that is incompatible with the Target interface.
3. **Adapter:** Implements the Target interface and translates the client's requests into calls to the Adaptee's interface.
4. **Client Code:** Interacts with the Target interface, unaware of the Adapter and Adaptee.

Real-Time Use Cases:

1. **Legacy System Integration:**
   - **Example:** Integrating a new system with an old legacy system that uses different interfaces.
   - **Reason:** The Adapter Pattern allows the new system to interact with the legacy system without modifying the legacy code.
2. **Third-Party Library Integration:**
   - **Example:** Using a third-party library that provides an API with a different interface than your application.
   - **Reason:** An adapter can be created to translate between your application's expected interface and the library's interface.
3. **Cross-Platform UI Development:**
   - **Example:** Creating a UI component that works on different platforms (e.g., Windows, macOS) with different underlying implementations.
   - **Reason:** The Adapter Pattern allows the UI component to interact with platform-specific implementations through a common interface.

Sample Program in Java:

Let's use an example of a simple media player application that needs to play different types of media files.

Scenario: We have a media player (`MediaPlayer`) that expects to play files via a `Media` interface. However, we have a legacy `LegacyAudioPlayer` class that plays audio but uses a different interface.

Objective: Create an adapter that allows `MediaPlayer` to work with `LegacyAudioPlayer`.

```
// Target Interface
interface Media {
    void play(String filename);
}

// Adaptee class
class LegacyAudioPlayer {
    void playAudio(String filename) {
        System.out.println("Playing audio file: " + filename);
    }
```

```
}

// Adapter class
class AudioAdapter implements Media {
    private LegacyAudioPlayer legacyAudioPlayer;

    public AudioAdapter(LegacyAudioPlayer legacyAudioPlayer) {
        this.legacyAudioPlayer = legacyAudioPlayer;
    }

    @Override
    public void play(String filename) {
        legacyAudioPlayer.playAudio(filename);
    }
}

// Client class
public class AdapterPatternDemo {
    public static void main(String[] args) {
        // Create an instance of the Adaptee
        LegacyAudioPlayer legacyAudioPlayer = new LegacyAudioPlayer();

        // Create an Adapter for the Adaptee
        Media mediaPlayer = new AudioAdapter(legacyAudioPlayer);

        // Client code uses the Target interface
        mediaPlayer.play("song.mp3");
    }
}
```

Explanation:

1. **Target Interface (`Media`):** Defines the `play(String filename)` method that the client expects.
2. **Adaptee (`LegacyAudioPlayer`):** Provides the functionality to play audio files but uses a different method (`playAudio`).
3. **Adapter (`AudioAdapter`):** Implements the `Media` interface and translates the `play` method call into a `playAudio` method call on the `LegacyAudioPlayer`.
4. **Client Code:** Uses the `Media` interface to interact with the `AudioAdapter`, which in turn interacts with the `LegacyAudioPlayer`.

Facade Design Pattern

Description:

The **Facade Design Pattern** is a structural design pattern that provides a simplified interface to a complex subsystem. It defines a higher-level interface that makes the subsystem easier to use by hiding the complexities and interactions between the subsystem components. The

main goal of the Facade Pattern is to provide a unified interface that makes a set of interfaces easier to understand and work with.

Key Components:

1. **Facade:** The simplified interface that clients use to interact with the subsystem.
2. **Subsystem Classes:** The classes that perform the actual work. These classes are complex and have intricate interactions, but their complexity is hidden behind the facade.

How It Works:

1. **Define the Facade:** The facade provides simple methods that delegate calls to the appropriate subsystem classes.
2. **Clients Use the Facade:** Clients interact with the facade, which internally coordinates with the subsystem classes to fulfill the request.

Real-Time Use Cases:

1. **Library Management System:**
   - **Example:** A library system with multiple subsystems for book management, user management, and loan management.
   - **Reason:** The facade can provide a simplified interface for common library operations like borrowing a book, returning a book, or registering a user.
2. **Home Automation System:**
   - **Example:** A home automation system with subsystems for lights, climate control, and security.
   - **Reason:** The facade can provide a single interface for controlling various home automation features, making it easier for users to operate.
3. **E-commerce System:**
   - **Example:** An e-commerce platform with subsystems for product catalog, order processing, payment processing, and shipping.
   - **Reason:** The facade can provide a unified interface for placing orders, making payments, and tracking shipments, simplifying the user experience.

Sample Program in Java:

Let's use an example of a home theater system with multiple subsystems for DVD player, projector, amplifier, and lights.

Scenario: The home theater system involves multiple components that need to be turned on/off and configured in a specific order. The facade will simplify this process for the user.

```
// Subsystem classes
class DVDPlayer {
    public void on() {
        System.out.println("DVD Player is on.");
    }

    public void play(String movie) {
        System.out.println("Playing movie: " + movie);
```

```
    }

    public void off() {
        System.out.println("DVD Player is off.");
    }
}

class Projector {
    public void on() {
        System.out.println("Projector is on.");
    }

    public void off() {
        System.out.println("Projector is off.");
    }
}

class Amplifier {
    public void on() {
        System.out.println("Amplifier is on.");
    }

    public void setVolume(int volume) {
        System.out.println("Setting volume to: " + volume);
    }

    public void off() {
        System.out.println("Amplifier is off.");
    }
}

class Lights {
    public void dim(int level) {
        System.out.println("Lights are dimmed to: " + level + "%");
    }
}

// Facade class
class HomeTheaterFacade {
    private DVDPlayer dvdPlayer;
    private Projector projector;
    private Amplifier amplifier;
    private Lights lights;

    public HomeTheaterFacade(DVDPlayer dvdPlayer, Projector projector,
Amplifier amplifier, Lights lights) {
        this.dvdPlayer = dvdPlayer;
        this.projector = projector;
        this.amplifier = amplifier;
        this.lights = lights;
    }

    public void watchMovie(String movie) {
        System.out.println("Get ready to watch a movie...");
        lights.dim(10);
        projector.on();
        amplifier.on();
```

```
        amplifier.setVolume(5);
        dvdPlayer.on();
        dvdPlayer.play(movie);
    }

    public void endMovie() {
        System.out.println("Shutting down movie theater...");
        dvdPlayer.off();
        amplifier.off();
        projector.off();
        lights.dim(100);
    }
}

// Main class to test the Facade pattern
public class FacadePatternDemo {
    public static void main(String[] args) {
        DVDPlayer dvdPlayer = new DVDPlayer();
        Projector projector = new Projector();
        Amplifier amplifier = new Amplifier();
        Lights lights = new Lights();

        HomeTheaterFacade homeTheater = new HomeTheaterFacade(dvdPlayer,
projector, amplifier, lights);

        homeTheater.watchMovie("Inception");
        System.out.println();
        homeTheater.endMovie();
    }
}
```

Explanation:

1. **Subsystem Classes:** Each subsystem class (`DVDPlayer`, `Projector`, `Amplifier`, `Lights`) has its own methods to perform specific actions.
2. **Facade Class (`HomeTheaterFacade`):** Provides a simplified interface for using the home theater system. It coordinates the actions of the subsystem classes to perform higher-level operations.
3. **Client Code:** Uses the facade to interact with the home theater system, making the process of watching and ending a movie straightforward.

Observer Design Pattern

The **Observer Design Pattern** is a behavioral design pattern in which an object, known as the subject, maintains a list of its dependents, called observers, and notifies them of any state changes, usually by calling one of their methods. This pattern is used to establish a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Key Components:

1. **Subject:** The object that holds the state and sends notifications to observers.
2. **Observer:** The object that needs to be informed about changes in the subject.
3. **ConcreteSubject:** A concrete implementation of the Subject that maintains the state and notifies observers.
4. **ConcreteObserver:** A concrete implementation of the Observer that reacts to changes in the subject's state.

How It Works:

1. **Define the Subject Interface:** This interface includes methods to attach, detach, and notify observers.
2. **Define the Observer Interface:** This interface includes an update method that gets called when the subject's state changes.
3. **ConcreteSubject:** Implements the Subject interface and maintains the state.
4. **ConcreteObserver:** Implements the Observer interface and updates its state based on the subject's state.

Real-Time Use Cases:

1. **Weather Stations:**
   - **Example:** A weather station notifies multiple displays (temperature, humidity, pressure) when there are changes in weather data.
2. **Stock Market Tickers:**
   - **Example:** A stock market tracker notifies multiple subscribers about stock price updates.
3. **Event Handling in GUI Applications:**
   - **Example:** Button clicks, text input changes, and other UI events notify listeners about changes.

Sample Program in Java:

Let's use a simple example of a `NewsAgency` (subject) that notifies its subscribers (observers) about news updates.

Scenario: A `NewsAgency` publishes news updates, and multiple `NewsChannel` objects (subscribers) receive the updates.

Objective: Use the Observer Design Pattern to notify multiple news channels whenever the news is updated in the `NewsAgency`.

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

// Observer interface
interface Observer {
    void update(String news);
}

// Subject class
```

```
class NewsAgency {
    private List<Observer> observers = new ArrayList<>();
    private String news;

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    public void setNews(String news) {
        this.news = news;
        notifyObservers();
    }

    private void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(news);
        }
    }
}

// Concrete Observer class
class NewsChannel implements Observer {
    private String news;

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

    public void display() {
        System.out.println("News Channel received news: " + news);
    }
}

// Main class to test the Observer pattern
public class ObserverPatternDemo {
    public static void main(String[] args) {
        NewsAgency newsAgency = new NewsAgency();
        NewsChannel channel1 = new NewsChannel();
        NewsChannel channel2 = new NewsChannel();

        newsAgency.addObserver(channel1);
        newsAgency.addObserver(channel2);

        newsAgency.setNews("Breaking News: Observer Pattern in Java!");
        newsAgency.setNews("Another Update: Learn Design Patterns
Easily!");
    }
}
```

Explanation:

1. **Observer Interface (`Observer`):** Defines the `update` method that will be called when the subject's state changes.
2. **Subject Class (`NewsAgency`):** Maintains a list of observers and notifies them of changes.
3. **Concrete Observer Class (`NewsChannel`):** Implements the `Observer` interface and updates its state based on the subject's state.
4. **Client Code:** Creates instances of `NewsAgency` (subject) and `NewsChannel` (observers). When the news is updated in the `NewsAgency`, the news channels are notified and display the news.

Chain of Responsibility Design Pattern:

The **Chain of Responsibility Design Pattern** is a behavioral design pattern that allows an object to send a command without knowing which object will handle the request. This pattern creates a chain of receiver objects. When a request is sent, it travels along the chain until it reaches an object that can handle it.

Key Components:

1. **Handler Interface/Abstract Class:** Defines an interface for handling requests and optionally implementing the link to the next handler.
2. **Concrete Handlers:** Implement the handler interface and handle requests they are responsible for. They can pass the request along the chain if they cannot handle it.
3. **Client:** Initiates the request to the chain of handlers.

How It Works:

1. **Client Sends Request:** The client creates a request and sends it to the first handler in the chain.
2. **Handlers Process Request:** Each handler decides either to process the request or to pass it to the next handler in the chain.
3. **Request is Handled:** The request is either processed by a handler or reaches the end of the chain without being processed.

Real-Time Use Cases:

1. **Technical Support System:**
    - **Example:** A technical support system where requests are passed through different levels of support (Level 1, Level 2, Level 3).
    - **Reason:** Each level handles the requests it is capable of, passing higher complexity issues up the chain.
2. **Logging Framework:**
    - **Example:** A logging framework that processes log messages through different loggers (console logger, file logger, database logger).
    - **Reason:** Each logger can process the message if it meets certain criteria and pass it along the chain otherwise.
3. **Event Handling in GUI Frameworks:**

- **Example:** Event handling where an event is passed through a chain of handlers (e.g., window, panel, button).
- **Reason:** Each handler processes events it is interested in and forwards the rest up the chain.

Sample Program in Java:

Let's create an example of a logging framework with different levels of logging: `InfoLogger`, `DebugLogger`, and `ErrorLogger`.

```
// Abstract Handler
abstract class Logger {
    public static int INFO = 1;
    public static int DEBUG = 2;
    public static int ERROR = 3;

    protected int level;
    protected Logger nextLogger;

    public void setNextLogger(Logger nextLogger) {
        this.nextLogger = nextLogger;
    }

    public void logMessage(int level, String message) {
        if (this.level <= level) {
            write(message);
        }
        if (nextLogger != null) {
            nextLogger.logMessage(level, message);
        }
    }

    abstract protected void write(String message);
}

// Concrete Handlers
class InfoLogger extends Logger {
    public InfoLogger(int level) {
        this.level = level;
    }

    @Override
    protected void write(String message) {
        System.out.println("Info Logger: " + message);
    }
}

class DebugLogger extends Logger {
    public DebugLogger(int level) {
        this.level = level;
    }

    @Override
    protected void write(String message) {
        System.out.println("Debug Logger: " + message);
```

```
    }
}

class ErrorLogger extends Logger {
    public ErrorLogger(int level) {
        this.level = level;
    }

    @Override
    protected void write(String message) {
        System.out.println("Error Logger: " + message);
    }
}

// Client code
public class ChainPatternDemo {
    private static Logger getChainOfLoggers() {
        Logger errorLogger = new ErrorLogger(Logger.ERROR);
        Logger debugLogger = new DebugLogger(Logger.DEBUG);
        Logger infoLogger = new InfoLogger(Logger.INFO);

        infoLogger.setNextLogger(debugLogger);
        debugLogger.setNextLogger(errorLogger);

        return infoLogger;
    }

    public static void main(String[] args) {
        Logger loggerChain = getChainOfLoggers();

        loggerChain.logMessage(Logger.INFO, "This is an information.");
        loggerChain.logMessage(Logger.DEBUG, "This is a debug level
information.");
        loggerChain.logMessage(Logger.ERROR, "This is an error
information.");
    }
}
```

Explanation:

1. **Abstract Handler (`Logger`):** Defines the interface for handling requests and managing the next handler in the chain.
2. **Concrete Handlers (`InfoLogger`, `DebugLogger`, `ErrorLogger`):** Implement the abstract handler and provide the specific handling logic.
3. **Client Code:** Sets up the chain of loggers and sends log messages at different levels.