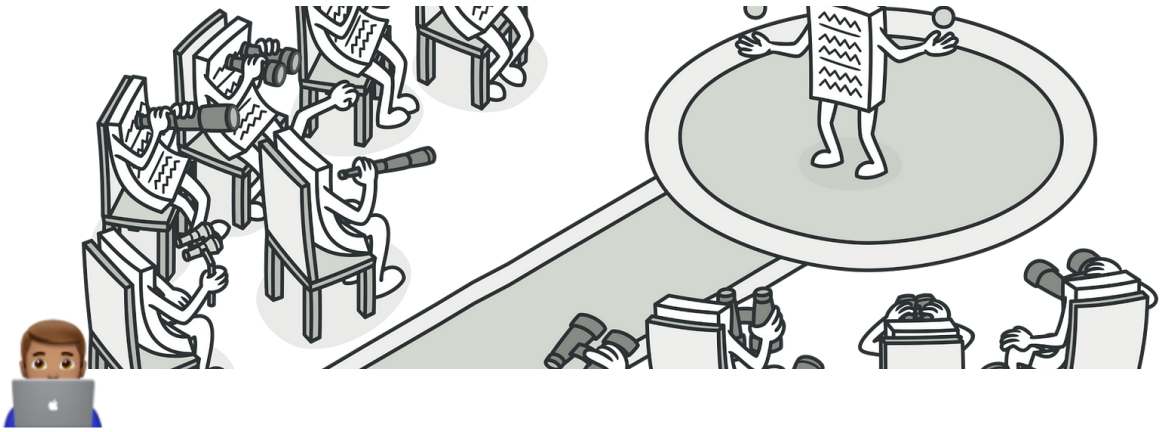
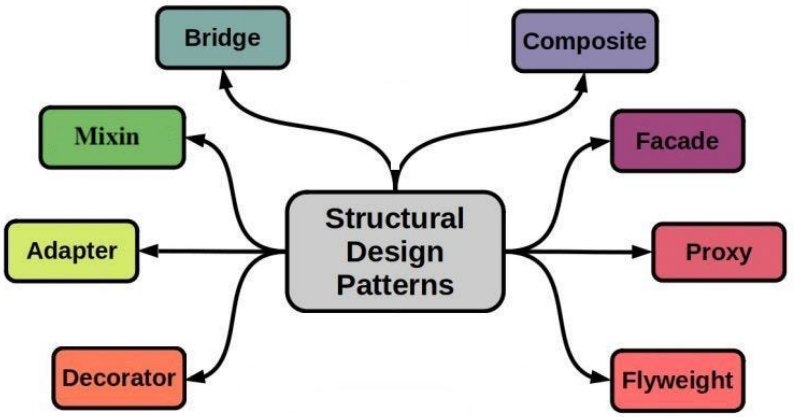


Design Patterns: Structural

▼ Structural Design Patterns:

- Structural patterns help organise different classes and objects to form larger, more efficient systems.
- These patterns are concerned with **how classes and objects are composed** to form larger structures while keeping the system **flexible and efficient**.
- The key focus is on **simplifying relationships** and **improving code flexibility** by changing how objects interact.

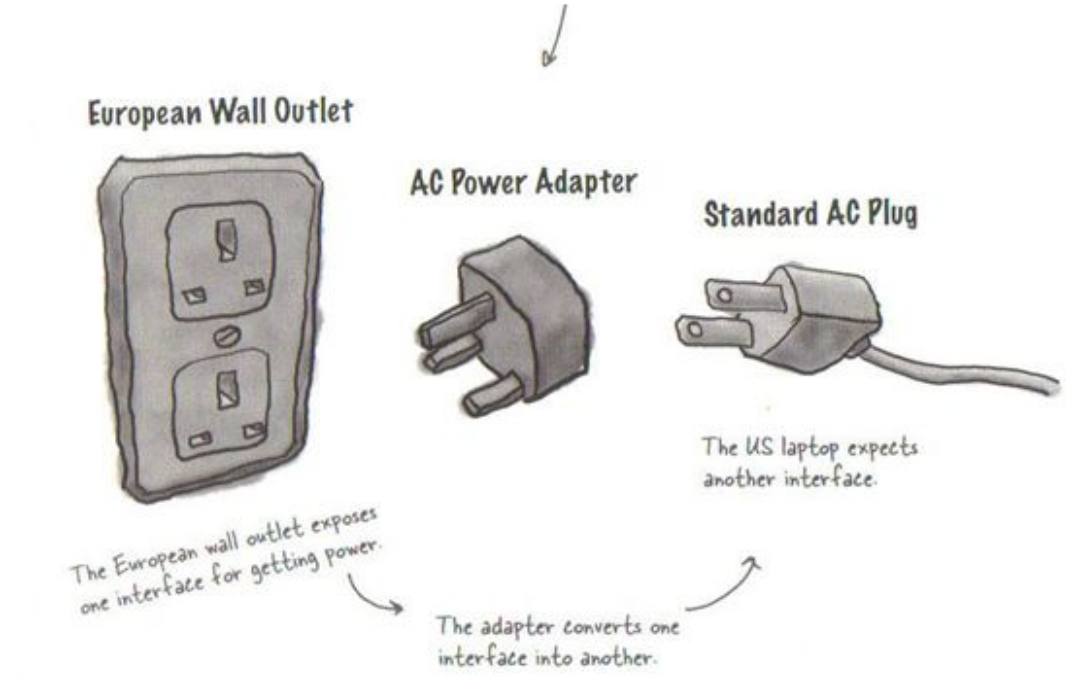


Core Concepts of Structural Patterns:

- Structural patterns promote composition (combining multiple objects) rather than extending classes through inheritance. This allows for more flexible and reusable systems.
- They promote reusing existing functionality through object composition and interfaces, reducing redundant code and improving system maintainability.

▼ What is the Adapter Pattern?

- The **Adapter Pattern** is a **structural design pattern** that allows objects with incompatible interfaces to collaborate.
- The pattern acts as a bridge between two incompatible interfaces, helping one object adapt to the interface expected by another object.
- It is commonly used when integrating legacy code or when you need to use a class that doesn't match the required interface.



Why do we need the Adapter Pattern?

- When you have an existing class that needs to interact with new classes, but their interfaces don't match.
- When integrating third-party libraries or frameworks with incompatible APIs.
- Allows existing, non-modifiable classes to fit into a new design without rewriting or heavily modifying existing code.

Adapter
Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.
https://refactoring.guru/design-patterns/adapter

The Adapter Pattern in Java | Baeldung
Understand the Adapter Design Pattern with a practical Java implementation
https://www.baeldung.com/java-adapter-pattern

▼ Basic Implementation of the Adapter Pattern

Here’s a simple example where we adapt an **AdvancedMediaPlayer** (which supports specific formats like VLC and MP4) to work with a **MediaPlayer** interface that supports generic media types.

Example: Media Player Adapter

```
// Target interface
interface MediaPlayer {
    void play(String audioType, String fileName);
}

// Adaptee interface with more specific functionali
ty
interface AdvancedMediaPlayer {
    void playVlc(String fileName);
    void playMp4(String fileName);
}

// Concrete implementation of AdvancedMediaPlayer f
or VLC
class VlcPlayer implements AdvancedMediaPlayer {
    @Override
    public void playVlc(String fileName) {
        System.out.println("Playing VLC file. Name:
" + fileName);
    }

    @Override
    public void playMp4(String fileName) {
        // Do nothing
    }
}

// Concrete implementation of AdvancedMediaPlayer f
or MP4
class Mp4Player implements AdvancedMediaPlayer {
    @Override
    public void playVlc(String fileName) {
        // Do nothing
```

```
    }

    @Override
    public void playMp4(String fileName) {
        System.out.println("Playing MP4 file. Name:
" + fileName);
    }
}

// Adapter class to bridge MediaPlayer with Advance
dMediaPlayer
class MediaAdapter implements MediaPlayer {

    AdvancedMediaPlayer advancedMusicPlayer;

    public MediaAdapter(String audioType) {
        if (audioType.equalsIgnoreCase("vlc")) {
            advancedMusicPlayer = new VlcPlayer();
        } else if (audioType.equalsIgnoreCase("mp
4")) {
            advancedMusicPlayer = new Mp4Player();
        }
    }

    @Override
    public void play(String audioType, String fileN
ame) {
        if (audioType.equalsIgnoreCase("vlc")) {
            advancedMusicPlayer.playVlc(fileName);
        } else if (audioType.equalsIgnoreCase("mp
4")) {
            advancedMusicPlayer.playMp4(fileName);
        }
    }
}

// Concrete class that uses the adapter
class AudioPlayer implements MediaPlayer {
```

```
    MediaAdapter mediaAdapter;

    @Override
    public void play(String audioType, String fileN
ame) {
        if (audioType.equalsIgnoreCase("mp3")) {
            System.out.println("Playing MP3 file. N
ame: " + fileName);
        } else if (audioType.equalsIgnoreCase("vl
c") || audioType.equalsIgnoreCase("mp4")) {
            mediaAdapter = new MediaAdapter(audioTy
pe);
            mediaAdapter.play(audioType, fileName);
        } else {
            System.out.println("Invalid media. " +
audioType + " format not supported");
        }
    }
}

// Client code
public class AdapterPatternDemo {
    public static void main(String[] args) {
        AudioPlayer audioPlayer = new AudioPlayer
();
        audioPlayer.play("mp3", "song.mp3"); // Out
put: Playing MP3 file. Name: song.mp3
        audioPlayer.play("mp4", "video.mp4"); // Ou
tput: Playing MP4 file. Name: video.mp4
        audioPlayer.play("vlc", "movie.vlc"); // Ou
tput: Playing VLC file. Name: movie.vlc
        audioPlayer.play("avi", "movie.avi"); // Ou
tput: Invalid media. avi format not supported
    }
}
```

▼ Key Points about the Adapter Pattern

- The Adapter Pattern is used when you want to use a class with an interface that isn't compatible with what you expect.
- The adapter separates the **interface logic** from the core functionality of the incompatible class.
- **Class Adapter vs Object Adapter**:
  - **Class Adapter** uses **inheritance** to adapt one interface to another.
    - **Example of Class Adapter:**
      - In the **Class Adapter** pattern, we use **inheritance** to adapt one interface to another. It works by creating an adapter class that inherits from the target class and adapts it to the required interface.
      - Here we have an example where a class adapter is used to adapt a **TemperatureSensor** interface to provide temperature in Celsius, while the adaptee returns temperature in Fahrenheit.

```
// Target interface (what the client ex
pects)
interface TemperatureSensor {
    double getTemperatureInCelsius();
}

// Adaptee class (returns temperature i
n Fahrenheit)
class FahrenheitSensor {
    public double getTemperatureInFahre
nheit() {
        return 98.6; // Fahrenheit valu
e
    }
}

// Class Adapter using inheritance
class TemperatureAdapter extends Fahren
heitSensor implements TemperatureSensor
```

```
{
    @Override
    public double getTemperatureInCelsi
us() {
        // Convert Fahrenheit to Celsiu
s
        return (getTemperatureInFahrenh
eit() - 32) * 5 / 9;
    }
}

// Client code
public class ClassAdapterDemo {
    public static void main(String[] ar
gs) {
        TemperatureSensor sensor = new
TemperatureAdapter();
        System.out.println("Temperature
in Celsius: " + sensor.getTemperatureIn
Celsius());
    }
}
```

- **Object Adapter** uses **composition** to wrap an object and adapt its interface.
  - **Example of Object Adapter:**
    - In the **Object Adapter** pattern, we use **composition** to wrap an instance of the adaptee class inside the adapter. The adapter forwards requests to the adaptee to make it compatible with the target interface.
    - Here we use composition to adapt a **FahrenheitSensor** to a **TemperatureSensor** interface. This is often considered more flexible than the class adapter because the adapter class doesn't need to inherit from the adaptee class.

```
// Target interface (what the client ex
pects)
interface TemperatureSensor {
    double getTemperatureInCelsius();
}

// Adaptee class (returns temperature i
n Fahrenheit)
class FahrenheitSensor {
    public double getTemperatureInFahre
nheit() {
        return 98.6; // Fahrenheit valu
e
    }
}

// Object Adapter using composition
class TemperatureAdapter implements Tem
peratureSensor {
    private FahrenheitSensor fahrenheit
Sensor;

    public TemperatureAdapter(Fahrenhei
tSensor fahrenheitSensor) {
        this.fahrenheitSensor = fahrenh
eitSensor;
    }

    @Override
    public double getTemperatureInCelsi
us() {
        // Convert Fahrenheit to Celsiu
s
        return (fahrenheitSensor.getTem
peratureInFahrenheit() - 32) * 5 / 9;
    }
}
```

```
// Client code
public class ObjectAdapterDemo {
    public static void main(String[] ar
gs) {
        FahrenheitSensor fahrenheitSens
or = new FahrenheitSensor();
        TemperatureSensor sensor = new
TemperatureAdapter(fahrenheitSensor);
        System.out.println("Temperature
in Celsius: " + sensor.getTemperatureIn
Celsius());
    }
}
```

▼ Advantages of Adapter Pattern

1. The Adapter Pattern lets you use existing classes (even those from third-party libraries) without modifying their source code.
2. The client code remains loosely coupled to the adaptee, as it doesn't know about its concrete implementation.
3. By introducing adapters, the logic of adapting one interface to another is encapsulated in a separate class, following the **Single Responsibility Principle**.

▼ Common Pitfalls in Adapter Pattern

- **Overuse of Adapter Pattern**
  - Developers sometimes use the Adapter Pattern when it isn’t needed, creating more layers of abstraction than necessary, which can increase the complexity of the code.

  Example: Unnecessary Adapter

  Let’s say you have a class `SquarePeg` that fits into a `SquareHole`. However, the developer creates an adapter to fit it into a `RoundHole` even though a simpler conversion or modification of the method could do the job.

  ```
  // Adaptee class
  class SquarePeg {
  ```

```
    private double width;

    public SquarePeg(double width) {
        this.width = width;
    }

    public double getWidth() {
        return width;
    }
}

// Target class
class RoundHole {
    private double radius;

    public RoundHole(double radius) {
        this.radius = radius;
    }

    public boolean fits(RoundPeg peg) {
        return this.radius >= peg.getRadius();
    }
}

// Unnecessary adapter
class SquarePegAdapter extends RoundPeg {
    private SquarePeg squarePeg;

    public SquarePegAdapter(SquarePeg squarePeg)
{
        this.squarePeg = squarePeg;
    }

    @Override
    public double getRadius() {
        // Over-complicating the logic
        return Math.sqrt(Math.pow((squarePeg.get
Width() / 2), 2) * 2);
```

```
    }
}
```

In this case, using the Adapter pattern introduces **unnecessary complexity** when a simple utility method to convert square pegs to round pegs would suffice.

Solution:

- Instead of using an adapter, consider refactoring the code to either modify the class to work directly or write a simple helper method.
- Overuse of design patterns can increase cognitive load and make the system harder to maintain.

- **Performance Overhead**
  - Every call to an adapter method is an additional function invocation, which can add **overhead** in scenarios where performance is critical (e.g., real-time systems).

Example: Adapter in Real-Time System

Imagine an adapter that wraps around a video processing system where performance is critical. Adding an unnecessary adapter layer can introduce latency.

```
// Target interface (expected by real-time syste
m)
interface VideoProcessor {
    void processVideo(String video);
}

// Adaptee class (legacy video processor)
class LegacyVideoProcessor {
    public void runVideo(String video) {
        System.out.println("Processing video usi
ng legacy system: " + video);
    }
}
```

```
// Adapter class that adds overhead
class VideoProcessorAdapter implements VideoProc
essor {
    private LegacyVideoProcessor legacyProcesso
r;

    public VideoProcessorAdapter(LegacyVideoProc
essor legacyProcessor) {
        this.legacyProcessor = legacyProcessor;
    }

    @Override
    public void processVideo(String video) {
        legacyProcessor.runVideo(video);  // Add
itional function call adds overhead
    }
}
```

In performance-critical applications like **real-time video processing**, the **additional call** to the adapter's `processVideo` method introduces unnecessary delay.

Solution:

- When performance is crucial, avoid using the Adapter pattern unless absolutely necessary.
- Instead, refactor the code to directly integrate the functionality, minimising function calls and keeping the design simple.

- **Tight Coupling**
  - Adapters can sometimes lead to **tight coupling** between components, especially if not designed carefully.
  - This makes it difficult to modify or extend the system, as changes in the adaptee might require corresponding changes in the adapter.

  Example: Tight Coupling with Adapter

Imagine an adapter that adapts a legacy `PaymentProcessor` to a modern `PaymentGateway` interface. If the legacy system undergoes changes, the adapter may need constant updates, leading to tight coupling between the adapter and the adaptee.

```
// Target interface (modern system)
interface PaymentGateway {
    void processPayment(double amount);
}

// Adaptee class (legacy system)
class LegacyPaymentProcessor {
    public void makePayment(double amount) {
        System.out.println("Processing payment o
f: $" + amount + " through legacy system");
    }
}

// Adapter class (tight coupling with the legacy
system)
class PaymentGatewayAdapter implements PaymentGa
teway {
    private LegacyPaymentProcessor legacyProcess
or;

    public PaymentGatewayAdapter(LegacyPaymentPr
ocessor legacyProcessor) {
        this.legacyProcessor = legacyProcessor;
    }

    @Override
    public void processPayment(double amount) {
        legacyProcessor.makePayment(amount);
    }
}
```

If the **LegacyPaymentProcessor** is updated, you might need to update the `PaymentGatewayAdapter` as well, making the system **fragile**

and harder to extend.

Solution:

- Design the adapter to minimise changes to both the adaptee and target interface.
- Use dependency injection to decouple the adapter from the specific implementation, which will reduce the tight coupling.

- **Complexity in Maintenance**
  - If too many adapters are used in the system, it can become difficult to track how classes and systems are interconnected.
  - This can increase the maintenance cost of the system and make debugging difficult.

  Example: Multiple Adapters

  Consider a system where multiple legacy systems are being adapted to fit a new architecture. Each legacy component requires a different adapter, leading to a complex web of adapters.

```
class LegacyComponentA {
    public void runA() {}
}

class LegacyComponentB {
    public void runB() {}
}

class AdapterA {
    private LegacyComponentA legacyA;
    public void execute() {
        legacyA.runA();
    }
}

class AdapterB {
    private LegacyComponentB legacyB;
    public void execute() {
        legacyB.runB();
```

```
    }
}
```

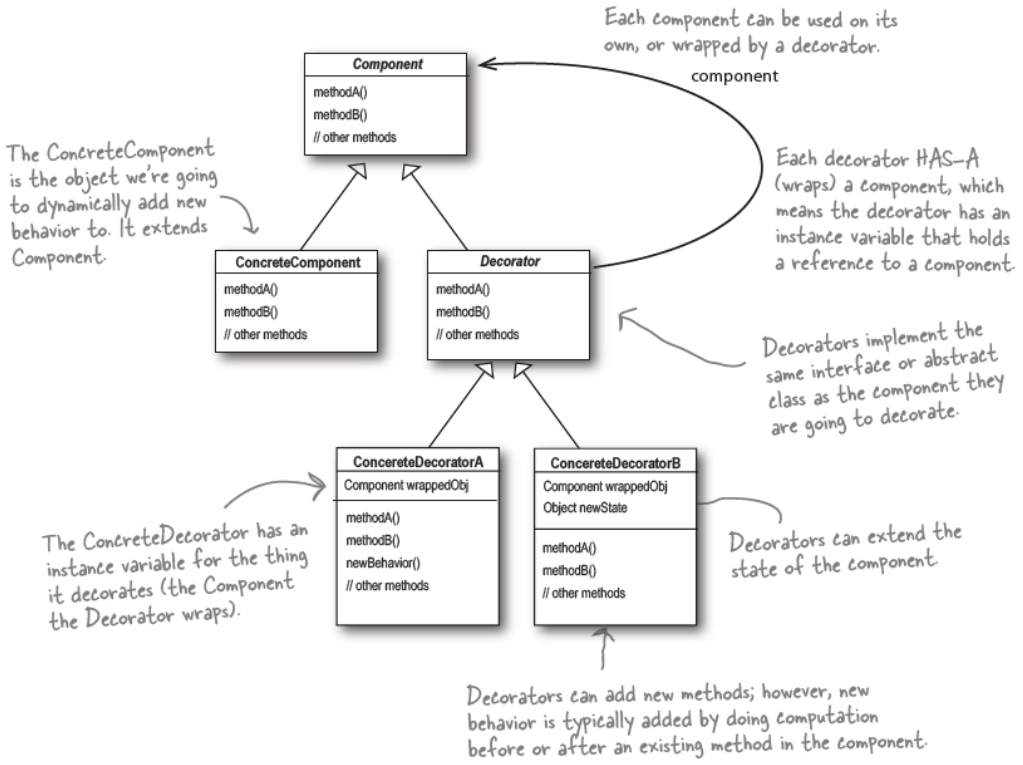
Here, you have multiple adapters for different components, which can become difficult to maintain when more components and adapters are added.

Solution:

- Keep the system simple by limiting the number of adapters.
- If multiple systems need to be adapted, consider consolidating adapters or using a **Facade Pattern** to manage multiple adapters behind a simpler interface.

▼ What is the Decorator Pattern?

- The **Decorator Pattern** allows for the dynamic addition of behaviours or responsibilities to individual objects without modifying the underlying class.
- This pattern is especially useful when subclassing would lead to an explosion of combinations of different behaviours.
- Instead of creating multiple subclasses, the **Decorator** pattern offers a more flexible approach.



Why use the Decorator Pattern?

- It provides a flexible alternative to subclassing for extending functionality.
- Instead of modifying a class directly, the pattern allows adding functionality dynamically at runtime.
- Examples:
  - Adding new features to a graphical user interface (e.g., scrollbars, borders).
  - Enhancing an object's functionality in a transparent manner.

Benefits of the Decorator Pattern

1. By using the decorator, you can divide functionality into small, focused classes. Each decorator class focuses on a specific behavior.
2. Classes can be extended without modifying the original code.
3. Rather than using inheritance to extend behavior, decorators use composition to wrap existing objects and extend their functionality.

Decorator

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain

https://refactoring.guru/design-patterns/decorator

The Decorator Pattern in Java | Baeldung

A guide to the decorator design pattern and its Java implementation

https://www.baeldung.com/java-decorator-pattern

▼ Basic Structure of the Decorator Pattern

1. **Component Interface** — Defines the base interface for objects that can have decorators.
2. **Concrete Component** — The class that is being decorated (i.e., the original object).

3. **Decorator Class** — Implements the same interface as the component and holds a reference to a component object.
4. **Concrete Decorator** — Implements additional functionality by calling methods on the wrapped object and adding new functionality before or after those calls.

Example: Basic Coffee

In this structure, the decorators wrap around the original component object, adding new behaviour:

- **Component Interface**: `Coffee`
- **Concrete Component**: `SimpleCoffee`
- **Decorator**: `CoffeeDecorator`
- **Concrete Decorators**: `MilkDecorator`, `SugarDecorator`

Let's consider a scenario where we want to dynamically add behaviours (like adding milk or sugar) to a coffee object.

1. **Component Interface**:

```
public interface Coffee {
    String getDescription();
    double cost();
}
```

2. **Concrete Component (Base Class)**:

```
public class SimpleCoffee implements Coffee {

    @Override
    public String getDescription() {
        return "Simple Coffee";
    }

    @Override
    public double cost() {
        return 2.00;  // Basic coffee cost
```

```
    }
}
```

3. **Decorator Class**:

```
public abstract class CoffeeDecorator implements
Coffee {
    protected Coffee coffee;  // The component b
eing decorated

    public CoffeeDecorator(Coffee coffee) {
        this.coffee = coffee;
    }

    public String getDescription() {
        return coffee.getDescription();
    }

    public double cost() {
        return coffee.cost();
    }
}
```

4. **Concrete Decorators**:

```
public class MilkDecorator extends CoffeeDecorat
or {

    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Mil
k";
    }
```

```
    @Override
    public double cost() {
        return coffee.cost() + 0.50;  // Additio
nal cost for milk
    }
}

public class SugarDecorator extends CoffeeDecora
tor {

    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Suga
r";
    }

    @Override
    public double cost() {
        return coffee.cost() + 0.20;  // Additio
nal cost for sugar
    }
}
```

Usage Example:

```
public class CoffeeShop {
    public static void main(String[] args) {
        Coffee simpleCoffee = new SimpleCoffee
();
        System.out.println(simpleCoffee.getDescr
iption() + " $" + simpleCoffee.cost());

        // Add Milk to the coffee
        Coffee milkCoffee = new MilkDecorator(si
```

```
mpleCoffee);
        System.out.println(milkCoffee.getDescrip
tion() + " $" + milkCoffee.cost());

        // Add Sugar to the milk coffee
        Coffee milkAndSugarCoffee = new SugarDec
orator(milkCoffee);
        System.out.println(milkAndSugarCoffee.ge
tDescription() + " $" + milkAndSugarCoffee.cost
());
    }
}
```

Expected Output:

```
Simple Coffee $2.0
Simple Coffee, Milk $2.5
Simple Coffee, Milk, Sugar $2.7
```

▼ Advantages of Decorator Pattern

- Enhances the behaviour of individual objects without modifying the underlying class or affecting other objects.
- The class is closed for modification but open for extension, making it easy to add new functionality by creating new decorators **Adheres to Open-Closed Principle (OCP)**.
- Instead of extending a class, behaviour is added dynamically by composing with other decorators.

▼ Decorators vs Inheritance & Composite

- **Decorator vs. Inheritance**:
  - Decorators provide a more flexible way to add behaviors than inheritance, as decorators don't affect other objects of the same class.
- **Decorator vs. Composite**:
  - The decorator pattern is often compared to the composite pattern, but while both involve objects that reference other

objects, the **composite** is focused on representing hierarchies, whereas the **decorator** focuses on adding responsibilities.

- For example, a composite tree of UI components might include windows, panels, and buttons, while decorators might add behaviors (e.g., borders, scrolling) to each of these components.

```
// Base interface for GUI components
public interface Component {
    void draw();
}

// Concrete component: TextBox
public class TextBox implements Component {
    @Override
    public void draw() {
        System.out.println("Drawing a TextBo
x");
    }
}

// Base Decorator class
public abstract class ComponentDecorator impl
ements Component {
    protected Component component;

    public ComponentDecorator(Component compo
nent) {
        this.component = component;
    }

    @Override
    public void draw() {
        component.draw();
    }
}
```

```
// Concrete Decorator: Border
public class BorderDecorator extends Componen
tDecorator {
    public BorderDecorator(Component componen
t) {
        super(component);
    }

    @Override
    public void draw() {
        super.draw();
        System.out.println("Adding border");
    }
}

// Concrete Decorator: ScrollBar
public class ScrollBarDecorator extends Compo
nentDecorator {
    public ScrollBarDecorator(Component compo
nent) {
        super(component);
    }

    @Override
    public void draw() {
        super.draw();
        System.out.println("Adding scroll ba
r");
    }
}

// Client code
public class Demo {
    public static void main(String[] args) {
        Component textBox = new TextBox();

        // Add a border decorator
        Component borderedTextBox = new Borde
```

```
rDecorator(textBox);

        // Add a scroll bar decorator
        Component scrollableBorderedTextBox =
new ScrollBarDecorator(borderedTextBox);

        // Draw the component
        scrollableBorderedTextBox.draw();
        // Output: Drawing a TextBox
        //         Adding border
        //         Adding scroll bar
    }
}
```

▼ Common Pitfalls in the Decorator Pattern:

- **Complexity of Layers**
  - When too many decorators are combined, it can become difficult to follow the flow of execution, as behaviour is distributed across multiple layers.

Example:

Consider a coffee ordering system where we add several layers of decorators (Milk, Sugar, Cream, Mocha, etc.):

```
// Base interface
public interface Coffee {
    String getDescription();
    double cost();
}

// Concrete implementation
public class SimpleCoffee implements Coffee {
    @Override
    public String getDescription() {
        return "Simple Coffee";
    }
    @Override
    public double cost() {
```

```
        return 2.00;
    }
}

// Decorators
public abstract class CoffeeDecorator implements
Coffee {
    protected Coffee coffee;
    public CoffeeDecorator(Coffee coffee) {
        this.coffee = coffee;
    }
    @Override
    public String getDescription() {
        return coffee.getDescription();
    }
    @Override
    public double cost() {
        return coffee.cost();
    }
}

public class MilkDecorator extends CoffeeDecorat
or {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }
    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Mil
k";
    }
    @Override
    public double cost() {
        return coffee.cost() + 0.50;
    }
}

public class SugarDecorator extends CoffeeDecora
```

```
tor {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }
    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Suga
r";
    }
    @Override
    public double cost() {
        return coffee.cost() + 0.20;
    }
}

// Example of adding too many layers
public class CoffeeShop {
    public static void main(String[] args) {
        Coffee coffee = new SimpleCoffee();
        coffee = new MilkDecorator(coffee);
        coffee = new SugarDecorator(coffee);
        coffee = new MilkDecorator(coffee);
        coffee = new SugarDecorator(coffee);  //
Adding layers upon layers
        System.out.println(coffee.getDescription
() + " $" + coffee.cost());
    }
}
```

Problem:

- The flow of the program becomes harder to trace with each new decorator added.
- Debugging issues related to cost or description becomes complex since many decorators are calling one another in a layered manner.

- **Violation of Single Responsibility Principle (SRP)**

- The SRP states that a class should have only one reason to change.
- Misuse of decorators to add multiple responsibilities (like adding milk and sugar in one decorator) can result in a violation of this principle.

Example:

```
public class CoffeeWithMilkAndSugarDecorator ext
ends CoffeeDecorator {
    public CoffeeWithMilkAndSugarDecorator(Coffe
e coffee) {
        super(coffee);
    }
    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Mil
k, Sugar";
    }
    @Override
    public double cost() {
        return coffee.cost() + 0.70;  // Adds bo
th Milk and Sugar in a single decorator
    }
}
```

Problem:

- This decorator is responsible for **two things**: adding both milk and sugar. If we want to change the price of milk or sugar, we have to modify this class.
- This violates the **SRP** because a single class should not handle multiple responsibilities.

- **Performance Issues**
  - Each decorator adds a method call, and if you have many decorators, it can introduce performance overhead due to increased method calls.

- In performance-critical systems, like real-time applications, this can degrade system performance.

Example:

Let’s say we have a system where decorators are used to add various security checks to a request processing system. Each decorator adds a layer of logic for validation or filtering:

```
public interface RequestProcessor {
    void processRequest(String request);
}

// Concrete implementation
public class SimpleRequestProcessor implements R
equestProcessor {
    @Override
    public void processRequest(String request) {
        System.out.println("Processing request:
" + request);
    }
}

// Security decorator for input validation
public class ValidationDecorator implements Requ
estProcessor {
    private RequestProcessor processor;

    public ValidationDecorator(RequestProcessor
processor) {
        this.processor = processor;
    }

    @Override
    public void processRequest(String request) {
        if (isValid(request)) {
            processor.processRequest(request);
        } else {
            System.out.println("Invalid request:
" + request);
```

```
        }
    }

    private boolean isValid(String request) {
        // Assume some complex validation logic
here
        return request.length() > 0;
    }
}

// Another security decorator for logging
public class LoggingDecorator implements Request
Processor {
    private RequestProcessor processor;

    public LoggingDecorator(RequestProcessor pro
cessor) {
        this.processor = processor;
    }

    @Override
    public void processRequest(String request) {
        System.out.println("Logging request: " +
request);
        processor.processRequest(request);
    }
}

// Usage
public class SecuritySystem {
    public static void main(String[] args) {
        RequestProcessor processor = new SimpleR
equestProcessor();
        processor = new ValidationDecorator(proc
essor);
        processor = new LoggingDecorator(process
or);
```

```
        processor.processRequest("Process this r
equest");
    }
}
```

Problem:

- In performance-critical systems (e.g., real-time processing), each decorator adds an extra method call.
- If you have many decorators for tasks like logging, validation, and caching, the performance overhead can accumulate.
