

Be part of a better internet. [Get 20% off membership for a limited time](#)

Simplifying Structural Design Patterns in Java: Creating Flexible Software Architecture



Rizwanul Haque · [Follow](#)

Published in Dev Genius · 11 min read · May 1, 2024



737



2





Structural design patterns in Java play a vital role in organizing code and defining relationships between classes and objects. These patterns focus on composition, inheritance, and the way objects are assembled to form larger structures. By employing structural design patterns, developers can create code that is more flexible, scalable, and easier to maintain. In this article, we'll delve into some of the most commonly used structural design patterns in Java and explore their implementation and benefits.

In this comprehensive guide, we'll explore some of the most widely used structural design patterns in Java, breaking down each pattern into simple terms for beginners to understand.

1. Adapter Pattern:

The Adapter Design Pattern is a structural pattern that allows objects with incompatible interfaces to collaborate. It acts as a bridge between two incompatible interfaces, converting the interface of one class into another interface that clients expect. This pattern is particularly useful when integrating legacy or third-party code into an existing system, where the interfaces don't match.

Example Scenario:

Let's consider a scenario where we have an existing application that communicates with an external service to fetch weather data. However, the external service's API has changed, and our application's code is tightly coupled to the old API. We want to update our application to use the new API without making significant changes to our existing codebase.

Implementation with Adapter Pattern:

To solve this problem, we can use the Adapter Pattern. We'll create an adapter class that implements the interface our application expects while

internally delegating the calls to the new API.

```
// WeatherService interface representing the expected interface in our applicati
interface WeatherService {
    String getWeather();
}

// ExternalWeatherService representing the new API with a different interface
class ExternalWeatherService {
    String fetchWeatherData() {
        // Logic to fetch weather data from the new API
        return "Sunny";
    }
}

// Adapter class that implements the WeatherService interface
class WeatherServiceAdapter implements WeatherService {
    private ExternalWeatherService externalService;

    public WeatherServiceAdapter(ExternalWeatherService externalService) {
        this.externalService = externalService;
    }

    @Override
    public String getWeather() {
        // Convert the response from the new API to match the expected format
        String weatherData = externalService.fetchWeatherData();
        return "Today's weather is: " + weatherData;
    }
}
```

Usage:

Now, let's see how we can use the adapter in our application code:

```
public class Application {  
    public static void main(String[] args) {  
        // Create an instance of the new API  
        ExternalWeatherService externalService = new ExternalWeatherService();  
  
        // Create an adapter for the new API  
        WeatherService weatherServiceAdapter = new WeatherServiceAdapter(externalService);  
  
        // Use the adapter to fetch weather data  
        String weather = weatherServiceAdapter.getWeather();  
        System.out.println(weather);  
    }  
}
```

In this example, the Adapter Pattern allowed us to seamlessly integrate the new API into our existing application without modifying the application's code that expects the old interface. By using the adapter, we achieved loose coupling between our application and the external service, making it easier to maintain and update in the future.

When to Use the Adapter Pattern:

Use the Adapter Pattern when:

1. You need to integrate existing code or third-party libraries with incompatible interfaces into your system.
2. You want to decouple client code from the implementation details of external services or components.
3. You need to reuse existing classes that don't have the interface your client code expects.

Overall, the Adapter Pattern is a powerful tool for achieving interoperability and maintaining flexibility in software systems.

2. Decorator Pattern:

The Decorator Design Pattern is a powerful tool in the software developer's arsenal, especially when it comes to adding or modifying the behavior of objects dynamically. In this article, we'll explore the Decorator Pattern in Java, understand its structure, and discuss scenarios where it can be effectively used.

Understanding the Decorator Pattern

The Decorator Pattern is a structural design pattern that allows behavior to be added to individual objects dynamically, without altering their structure. It achieves this by creating a chain of decorators, each adding new functionalities to the original object. This pattern promotes code reusability and flexibility by enabling objects to be extended in a modular way.

Example Scenario: Coffee Shop

Let's consider a scenario of a coffee shop where customers can order different types of coffee (e.g., Espresso, Latte, Cappuccino) with various toppings (e.g., milk, sugar, whipped cream). The challenge is to create a flexible system that allows customers to customize their coffee orders with different combinations of toppings.

Implementation with Decorator Pattern

First, we define an interface representing the Coffee:

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

Next, we create concrete implementations of the Coffee interface for different types of coffee:

```
public class Espresso implements Coffee {  
    @Override  
    public String getDescription() {  
        return "Espresso";  
    }  
  
    @Override  
    public double cost() {  
        return 1.99;  
    }  
}  
  
public class Latte implements Coffee {  
    @Override  
    public String getDescription() {  
        return "Latte";  
    }  
  
    @Override  
    public double cost() {  
        return 2.49;  
    }  
}
```

Now, let's define decorators for adding toppings:


```
public abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

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

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

    public double cost() {
        return decoratedCoffee.cost();
    }
}

public class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    public String getDescription() {
        return super.getDescription() + ", with Milk";
    }

    public double cost() {
        return super.cost() + 0.50;
    }
}
```

Client Code Usage

Now, let's see how we can use the Decorator Pattern in client code:

```
public class CoffeeShop {  
    public static void main(String[] args) {  
        // Order a plain Latte  
        Coffee latte = new Latte();  
        System.out.println("Ordered: " + latte.getDescription() + ", Cost: $" +  
  
        // Order a Latte with Milk  
        Coffee latteWithMilk = new MilkDecorator(new Latte());  
        System.out.println("Ordered: " + latteWithMilk.getDescription() + ", Cos  
    }  
}
```

When to Use Decorator Pattern

The Decorator Pattern is useful in the following scenarios:

- When you need to add or modify the behavior of objects dynamically at runtime.
- When you want to avoid the complexities of subclassing to extend functionality.

- When you need to add or remove responsibilities from individual objects without affecting other objects.

In conclusion, the Decorator Design Pattern in Java is a versatile tool for extending and customizing the behavior of objects in a flexible and modular way. By employing this pattern, developers can create code that is easier to maintain, extend, and understand, making it an essential pattern in modern software development.

3. Facade Pattern:

In software development, complex systems often consist of numerous interconnected components with intricate interactions. Managing this complexity can be challenging, especially when dealing with subsystems that have their own interfaces and dependencies. The Facade Design Pattern offers a solution by providing a unified interface to a set of interfaces in a subsystem, simplifying their usage for clients. In this article, we'll explore the Facade pattern in Java with a code example and discuss scenarios where it can be beneficial.

Understanding the Facade Pattern

The Facade Pattern is a structural design pattern that provides a simplified interface to a complex system of classes, subsystems, or libraries. It encapsulates the interactions between these components behind a single, easy-to-use interface, shielding clients from the complexities of the underlying implementation. By doing so, it promotes code readability, maintainability, and ease of use.

Example Scenario: Multimedia Player Library

Consider a multimedia player library that consists of various subsystems such as audio playback, video playback, and playlist management. Each subsystem has its own set of classes and interfaces, making it cumbersome for clients to interact with them directly.

Implementation with Facade Pattern

Let's create a Facade class, `MultimediaPlayer`, that provides a simplified interface to the multimedia player library:

```
class MediaPlayer {  
    private AudioPlayer audioPlayer;  
    private VideoPlayer videoPlayer;  
    private PlaylistManager playlistManager;  
  
    public MediaPlayer() {
```

```
this.audioPlayer = new AudioPlayer();
this.videoPlayer = new VideoPlayer();
this.playlistManager = new PlaylistManager();
}

public void playAudio(String audioFile) {
    audioPlayer.play(audioFile);
}

public void playVideo(String videoFile) {
    videoPlayer.play(videoFile);
}

public void createPlaylist(List<String> mediaFiles) {
    playlistManager.createPlaylist(mediaFiles);
}

public void playPlaylist() {
    playlistManager.play();
}
}
```

In this example, the `MultimediaPlayer` class acts as a facade for the subsystems: `AudioPlayer`, `VideoPlayer`, and `PlaylistManager`. It provides simplified methods for playing audio, playing video, creating playlists, and playing playlists, hiding the complexities of the underlying subsystems from clients.

Client Code Integration

Now, let's see how clients can use the `MultimediaPlayer` facade to interact with the multimedia player library:

```
public class Client {  
    public static void main(String[] args) {  
        MultimediaPlayer player = new MultimediaPlayer();  
  
        player.playAudio("song.mp3");  
        player.playVideo("movie.mp4");  
  
        List<String> playlist = Arrays.asList("song1.mp3", "song2.mp3", "song3.m  
        player.createPlaylist(playlist);  
        player.playPlaylist();  
    }  
}
```

When to Use the Facade Pattern

The Facade Pattern is suitable in the following scenarios:

1. **Complex Systems:** When dealing with complex systems composed of multiple subsystems, each with its own set of classes and interfaces.
2. **Simplifying Client Interfaces:** When you want to provide a simplified interface to a complex subsystem, shielding clients from its

implementation details.

3. **Legacy Systems Integration:** When integrating legacy systems or third-party libraries into modern applications, and you need to provide a more user-friendly interface.

The Facade Design Pattern simplifies the usage of complex systems by providing a unified interface to their subsystems. By encapsulating the complexities behind a facade, it promotes code maintainability, readability, and ease of use. When faced with complex systems or subsystems, consider using the Facade Pattern to streamline interactions and improve the overall design of your Java applications.

4. Composite Pattern:

The Composite Design Pattern is a structural design pattern that allows you to compose objects into tree structures to represent part-whole hierarchies. It enables clients to treat individual objects and compositions of objects uniformly, making it easier to work with complex structures. In this article, we'll explore the Composite Pattern in Java with a code example and discuss scenarios where it's most beneficial.

Understanding the Composite Pattern

The Composite Pattern consists of three key components:

1. **Component:** Represents the common interface for all objects in the composition, whether they are leaf nodes (individual objects) or composite nodes (collections of objects).
2. **Leaf:** Represents individual objects that have no children. They implement the Component interface.
3. **Composite:** Represents collections of objects that have child components. They also implement the Component interface and provide methods for adding, removing, and accessing child components.

Example Scenario: Representing a File System

Consider a scenario where we need to represent a hierarchical file system consisting of directories and files. Directories can contain both files and subdirectories, while files are leaf nodes that contain data. The Composite Pattern is well-suited for modeling such hierarchical structures.

Implementation with Composite Pattern

Let's implement the Composite Pattern to represent the file system:


```
interface FileSystemComponent {
    void printName();
}

class File implements FileSystemComponent {
    private String name;

    public File(String name) {
        this.name = name;
    }

    @Override
    public void printName() {
        System.out.println("File: " + name);
    }
}

class Directory implements FileSystemComponent {
    private String name;
    private List<FileSystemComponent> components;

    public Directory(String name) {
        this.name = name;
        components = new ArrayList<>();
    }

    public void addComponent(FileSystemComponent component) {
        components.add(component);
    }

    public void removeComponent(FileSystemComponent component) {
        components.remove(component);
    }

    @Override
```

```
public void printName() {  
    System.out.println("Directory: " + name);  
    for (FileSystemComponent component : components) {  
        component.printName();  
    }  
}
```

In this implementation, `FileSystemComponent` represents the common interface for both files and directories. `File` represents a leaf node, while `Directory` represents a composite node that can contain other components.

Client Code Usage

Now, let's see how we can use the Composite Pattern in client code:

```
public class Client {  
    public static void main(String[] args) {  
        Directory root = new Directory("Root");  
        Directory folder1 = new Directory("Folder 1");  
        Directory folder2 = new Directory("Folder 2");  
        File file1 = new File("File 1.txt");  
        File file2 = new File("File 2.txt");  
  
        root.addComponent(folder1);  
        root.addComponent(folder2);  
        folder1.addComponent(file1);  
        folder2.addComponent(file2);  
    }  
}
```

```
        root.printName();  
    }  
}
```

When to Use the Composite Pattern

- Use the Composite Pattern when you need to represent part-whole hierarchies, where individual objects and compositions of objects are treated uniformly.
- Use it to simplify the client code by allowing it to work with both individual objects and collections of objects in a consistent manner.
- Use it when you have recursive structures, such as directories and files, menus and menu items, or graphical shapes and their compositions.

The Composite Design Pattern is a powerful tool for representing hierarchical structures in a simple and unified manner. By employing this pattern, developers can build flexible and extensible systems that are easy to maintain and understand. Whether modeling file systems, organizational charts, or graphical user interfaces, the Composite Pattern provides a clear and elegant solution for managing complex hierarchies in Java applications.

5. Proxy Pattern:

The Proxy Design Pattern is a structural design pattern that provides a surrogate or placeholder for another object to control access to it. It allows for additional functionality to be provided when accessing an object, such as lazy initialization, access control, logging, or monitoring. In this article, we'll delve into the Proxy Pattern in Java, along with a code example, and discuss scenarios where it can be effectively used.

Understanding the Proxy Pattern

The Proxy Pattern involves three main components:

1. **Subject:** Defines the common interface for the RealSubject and Proxy so that the Proxy can be used anywhere the RealSubject is expected.
2. **Proxy:** Maintains a reference to the RealSubject and provides an identical interface as the RealSubject, allowing the Proxy to control access to the RealSubject and add additional functionality if needed.
3. **RealSubject:** Represents the real object that the Proxy is providing access to.

Example Scenario: Access Control with a Virtual Library

Imagine we have a virtual library system where users can access digital books. However, we want to restrict access to certain books based on the user's subscription level. In this scenario, we can use the Proxy Pattern to control access to the books based on the user's subscription level.

Implementation with Proxy Pattern

Let's create the components of our Proxy Pattern:

1. Subject (Book): Interface representing the common behavior of all books.

```
interface Book {  
    void display();  
}
```

2. RealSubject (RealBook): Represents the real book that users can access.

```
class RealBook implements Book {  
    private String title;  
  
    public RealBook(String title) {  
        this.title = title;  
    }  
}
```

```
}

@Override
public void display() {
    System.out.println("Displaying book: " + title);
}
}
```

3. Proxy (ProxyBook): Controls access to the RealSubject and adds functionality for access control.

```
class ProxyBook implements Book {
    private RealBook realBook;
    private String subscriptionLevel;

    public ProxyBook(String title, String subscriptionLevel) {
        this.realBook = new RealBook(title);
        this.subscriptionLevel = subscriptionLevel;
    }

    @Override
    public void display() {
        if (hasAccess()) {
            realBook.display();
        } else {
            System.out.println("You do not have access to this book.");
        }
    }
}
```

```
private boolean hasAccess() {  
    // Check user's subscription level  
    return subscriptionLevel.equals("premium");  
}  
}
```

Usage of Proxy Pattern

Now, let's see how we can use the Proxy Pattern to control access to books based on the user's subscription level:

```
public class Client {  
    public static void main(String[] args) {  
        Book book1 = new ProxyBook("Java Programming", "premium");  
        Book book2 = new ProxyBook("Design Patterns", "basic");  
  
        book1.display(); // Displaying book: Java Programming  
        book2.display(); // You do not have access to this book.  
    }  
}
```

When to Use Proxy Pattern

The Proxy Pattern is suitable in various scenarios, including:

1. **Lazy Initialization:** Delaying the creation of a costly object until it's actually needed.
2. **Access Control:** Controlling access to an object based on certain criteria, such as user permissions.
3. **Logging and Monitoring:** Adding logging or monitoring functionality without modifying the original object's code.
4. **Caching:** Storing the results of expensive operations and returning the cached result when the same operation is requested again.

The Proxy Design Pattern in Java provides a flexible and powerful way to control access to objects and add additional functionality when interacting with them. Whether it's for access control, lazy initialization, or logging, the Proxy Pattern allows developers to achieve better control and security in their applications. By understanding and applying the Proxy Pattern effectively, developers can enhance the robustness and maintainability of their software systems.

Thank you for reading until the end. Before you go:

- Please consider **clapping** and **following** me! 🙌

- Follow me on [X](#) | [LinkedIn](#)

Java

Design Patterns

Structural Design Pattern

Software Engineering



Written by Rizwanul Haque

1.1K Followers · Writer for Dev Genius

Follow



Lead Android Developer | Passionate about Coding | Sharing Insights 🚀 | Let's explore together! 📱💻 #AndroidDev #Kotlin #Tech

More from Rizwanul Haque and Dev Genius



 Rizwanul Haque in Stackademic

Important Coroutine Interview Questions for Experienced Andro...

Coroutines have revolutionized asynchronous programming in Kotlin, offering a powerful...

Feb 13  1K  3



Most Asked
Java 8
Interview
Coding Questions
(Part-1)


 Anusha SP in Dev Genius

Java 8 Coding and Programming Interview Questions and Answers

It has been 8 years since Java 8 was released. I have already shared the Java 8 Interview...

Jan 31, 2023  1K  15



 Amit Mishra in Dev Genius

45 JavaScript Super Hacks Every Developer Should Know



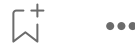
 Rizwanul Haque

SSL Pinning in Android Apps for Enhanced Security

JavaScript is a versatile and powerful language that is essential for modern web...

Introduction:

May 17 🖱 822 💬 11



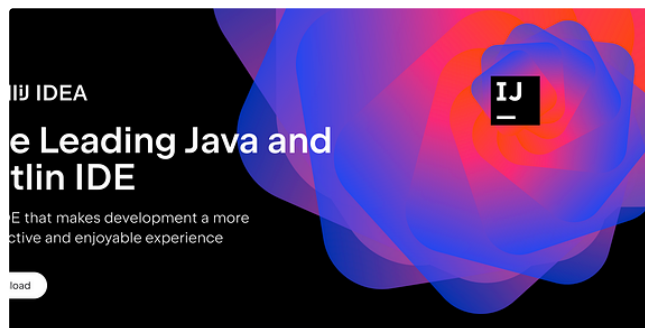
Dec 20, 2023 🖱 835 💬 2



See all from Rizwanul Haque

See all from Dev Genius

Recommended from Medium



 Gumparthy Pavan Kumar



 Renan Schmitt in JavaJams

IntelliJ IDE

If you are a developer and have not faced this issue yet, it is still worth reading, as at some...

Jun 8  38

Jun 24



176

 3

Lists



General Coding Knowledge

20 stories · 1360 saves



Stories to Help You Grow as a Software Developer

19 stories · 1184 saves



Leadership

51 stories · 372 saves



Good Product Thinking

11 stories · 622 saves





Learn to create a shared library in Spring Boot, enhancing code reusability, simplifyin...

 Jun 25
  244
  7



Failing Despite Preparation

5d ago 102



Mayank Sharma

A real life example to an interview question which is not Animal, Dog and cat classes.

 Jun 18
 70
 3



Hayk Simonyan in Level Up Coding

Prepare for system design interviews with this guide to designing a WhatsApp-like...

Jun 18  944  6



See more recommendations

[Help](#) [Status](#) [About](#) [Careers](#) [Press](#) [Blog](#) [Privacy](#) [Terms](#) [Text to speech](#) [Teams](#)