Structural Design Patterns

> *"I am wary of the whole dreary deadening structured mess that we have built into such a glittering top-heavy structure that there is nothing left to see but the glitter, and the brute routines of maintaining it."* — John D. MacDonald

Structural design patterns *in software engineering focus on the composition and organization of classes and objects to create larger, more complex structures, utilizing the concept of inheritance to combine interfaces or implementations, allowing for greater flexibility and efficiency in the overall design.*

Creational Design Patterns dealt with object creation mechanisms and we discussed various such patterns in this section, which help us to create objects and use them efficiently based on business requirements. But, till now we have only used simpler class structures and as we move towards complex class structures, we would be leveraging Structural Design Patterns. These patterns provide guidelines on how to create objects in a flexible and reusable manner, promoting decoupling and enhancing the scalability of our systems. Now, as we transition towards structural design patterns, our focus shifts from the creation of individual objects to the composition and organization of these objects to form larger structures. These patterns help us understand how to assemble classes and objects to build robust architectures that are adaptable, efficient, and maintainable. These patterns utilize **inheritance** and **composition** to create **interfaces** and **implementations** that enable the construction of flexible software systems.

Different business use cases will call for usage of different structural patterns implementation. Let us start with Adapter Patterns.

1.1. Adapter Pattern

Adapter Patterns *are used to allow classes work together, which would not have been possible due to incompatible interfaces. It converts interface of a class into another interface as expected by client.*

In Software industry, applications frequently encounter the need to seamlessly integrate with diverse third-party applications, enriching their functionality and enhancing user experiences. Moreover, as technology evolves and businesses strive to stay ahead, the process of modernization becomes inevitable.

Third-party applications may have their own unique interfaces, protocols, or data formats that differ from the expected interface of the application.

A crucial aspect of modernization involves addressing legacy behaviours and attributes. These aspects, which may have served the application well in the past, require thoughtful consideration and refinement to meet the evolving requirements of the present. The process involves implementing updated functionalities, optimizing performance, and adhering to the latest architectural patterns.

One of the key principles in software development is the idea of designing for **reusability**. However, there are situations where a toolkit class that is intended to be reusable may not fulfil its purpose because its interface doesn't align with the specific requirements of a particular application or domain.

Carrying on the vehicle example context, let us assume that a business requirement arises which demands conversion of Legacy Vehicle Data format into Modern Data Format. There might be changes in attribute names and how some of the attribute values are perceived in modern system. This is where Adapter Pattern will come handy.

Here, we have a legacy system data format which contains make, model, manufacturing year and engine used in a vehicle. However, in new data format, these attribute names are changed, and engine names have changed slightly. No doubt, legacy interface will not compatible with modern interface at compilation.

 Hence, to solve this, we implement an adapter between the two data formats, which encapsulates the data conversion logic from the client.

 Let us look at code snippets to understand better.

```
package Adapter;

public class LegacyVehicleData {
    private String make;
    private String model;
    private int manufacturingYear;
    private String engineModel;

    // getter and setter method implementation

package Adapter;

public class ModernVehicleData {
    private String brand;
    private String type;
    private int year;
    private String engineType;

    // getter and setter method implementation

package Adapter;

public interface EngineAdapter {
    String convertToModernEngineFormat();
}
package Adapter;

public interface LegacyVehicleDataAdapter {
    ModernVehicleData convertToModernFormat();
}
package Adapter;

public class EngineAdapterImpl implements EngineAdapter {
    private LegacyVehicleData legacyVehicleData;

    public EngineAdapterImpl(LegacyVehicleData legacyVehicleData) {
        this.legacyVehicleData = legacyVehicleData;
    }

    @Override
    public String convertToModernEngineFormat() {
        // Perform engine data conversion/transformation from legacy to modern format
        String engineModel = legacyVehicleData.getEngineModel();
        // Example conversion logic
        if (engineModel.equals("V4")) {
            return "Inline-4";
```

```
        } else if (engineModel.equals("V6")) {
            return "V6";
        } else {
            return "Unknown";
        }
    }

}

package Adapter;

/**
 * LegacyVehicleDataAdapterImpl
 */
public class LegacyVehicleDataAdapterImpl implements LegacyVehicleDataAdapter {
    private LegacyVehicleData legacyVehicleData;
    private EngineAdapter engineAdapter;

    public LegacyVehicleDataAdapterImpl(LegacyVehicleData legacyVehicleData) {
        this.legacyVehicleData = legacyVehicleData;
        this.engineAdapter = new EngineAdapterImpl(legacyVehicleData);
    }

    @Override
    public ModernVehicleData convertToModernFormat() {
        String brand = legacyVehicleData.getMake();
        String type = legacyVehicleData.getModel();
        int year = legacyVehicleData.getManufacturingYear();
        String engineType = engineAdapter.convertToModernEngineFormat();

        return new ModernVehicleData(brand, type, year, engineType);
    }

}
package Adapter;

// Client code
public class Client {
    public static void main(String[] args) {
        // Assume we have a legacy vehicle data instance
        LegacyVehicleData legacyData = new LegacyVehicleData("LegacyMake", "LegacyModel",
2000, "V4");

        // Create an adapter instance and convert legacy data to modern format
        LegacyVehicleDataAdapter adapter = new LegacyVehicleDataAdapterImpl(legacyData);
        ModernVehicleData modernData = adapter.convertToModernFormat();

        System.out.println("Engine details: ---" + modernData.getEngineType());
    }
}
```

 In the code above, we are using two data formats *LegaceVehicleData* and *ModerVehicleData.* Both classes have similar attributes with different names though. Client which has been using Legacy format yet, now leverages *LegacyVehicleDataAdapter* to map the data from older format to latest format. *LegacyVehicleDataAdapter* class encapsulates the logic from client code about conversion of data format and makes the code loosely coupled. We can further reuse this code in other parts of the application, which has been using older data format to migrate towards latest format. *LegacyVehicleDataAdapter* internally uses another adapter named *EngineAdapter* which encapsulates the logic of mapping older engine types to new types.

🎯 **Adapter pattern** allows for the reuse of existing classes or components that have incompatible interfaces. By creating adapters, these classes can be integrated into new systems without requiring modifications to their original code. It bridges the gap between disparate interfaces, ensuring that they can communicate and collaborate effectively. This promotes code reuse and avoids the need to duplicate functionality.

🚨 Following are some of the considerations we must keep in mind while using Adapter Pattern:

1. **Cost vs. Benefit**: When deciding whether to use an adapter, weigh the benefits it provides against the associated costs. If the incompatibility between interfaces is causing significant issues or hindering the development process, the benefits of using an adapter may outweigh the costs. However, if the differences are minimal or the integration is straightforward, an adapter may not be necessary.
2. **Performance Impact**: Adapters introduce an extra layer of indirection, which can potentially impact performance. The delegation from the adapter to the adapted object adds a small overhead in terms of execution time and memory. If performance is a critical concern in your specific use case, carefully evaluate the impact of using an adapter and consider alternatives if necessary.

1.2. Bridge Pattern

💡 ***Bridge Pattern*** *is used to decouple the abstraction and implementation hierarchies, enabling them to evolve independently. These hierarchies are then connected to each other via object composition, forming a bridge-like structure. This promotes loose coupling between the two, making it easier to modify or extend each hierarchy without affecting the other.*

It achieves decoupling of the abstraction and implementation hierarchies by creating a bridge between the abstraction (abstraction interface) and its concrete implementations (implementor interface). It allows for a flexible relationship between the abstraction and the implementation. The abstraction can use different implementations at runtime by referencing different concrete implementor objects. This enables dynamic switching or selection of implementations based on the requirements or configuration.

Okay, enough of talking in silos. Let us understand the use case in terms of a business scenario.

We are developing a messaging system app which aims to provide a flexible and extensible platform for sending messages across different channels, such as emails and SMS. The system needs to support various formatting options for the messages, such as plain text and HTML. The Bridge pattern is used to separate the message sending functionality from the formatting logic, enabling easy integration of new message formats and channels.

If we try to implement the above scenario in an intuitive approach, we may create a base class *MessageSender* which would be extended by SMS and Email Sender classes, and SMS sender would use Text Message Formatter and Email Sender would use HTML Message Formatter (see Figure 5-1). Although, at first look, there is nothing wrong with this approach, however at looking closely we find a tight coupling scenario between Text Message Formatter and SMS sender and similarly for Email Sender use case with HTML formatter.

🚨 This approach lacks flexibility as each sender class is directly responsible for both sending the message and formatting it. This means that any changes or additions to the formatting logic would

require modifications in each sender class, leading to code duplication and potential issues when maintaining or extending the system.

 **Bridge Pattern** promotes **composition over inheritance** and helps decouple high level functionality from low level implementation.

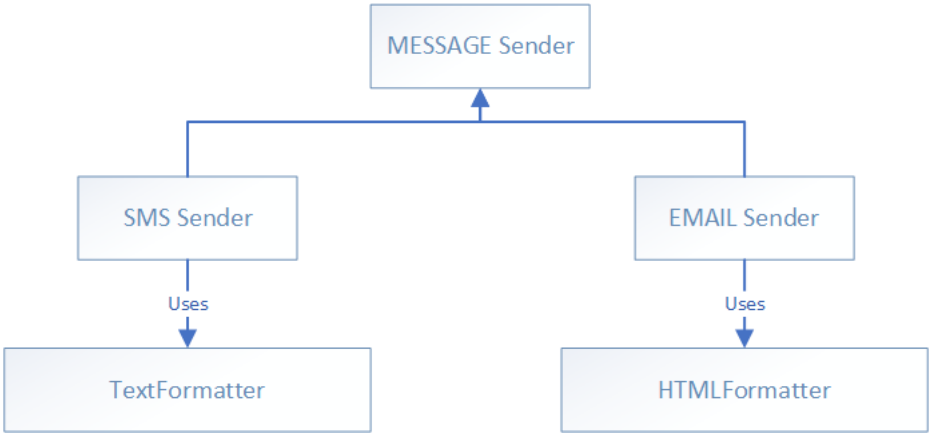

Figure 5-0-1

 Now, let us modify the architecture (see Figure 5-2) using **Bridge Pattern** and checkout following code example.

```
/**
 * This abstract class represents a message sender.
 */
public abstract class MessageSender {
    protected MessageFormatter formatter;

    /**
     * Constructs a MessageSender object with the given formatter.
     *
     * @param formatter the formatter to be used for formatting messages
     */
    public MessageSender(MessageFormatter formatter) {
        this.formatter = formatter;
    }

    /**
     * Sends a message.
     *
     * @param message the message to be sent
     */
    public abstract void sendMessage(String message);
}

public interface MessageFormatter {
    String formatMessage(String message);
}

public class HTMLFormatter implements MessageFormatter {
    @Override
    public String formatMessage(String message) {
        // Perform plain text formatting
        return " <html><body>" + message + "</body></html>";
    }
}

public class PlainTextFormatter implements MessageFormatter {
```

```
    @Override
    public String formatMessage(String message) {
        // Perform plain text formatting
        return "[Plain Text] " + message;
    }
}
/**
 * The  SMSSender  class is a concrete implementation of the  MessageSender  abstract class.
 * It is responsible for sending SMS messages using a specified message formatter.
 */
public class SmsSender extends MessageSender {
    public SmsSender(MessageFormatter formatter) {
        super(formatter);
    }

    @Override
    public void sendMessage(String message) {
        String formattedMessage = formatter.formatMessage(message);

        // Send SMS with the formatted message
        System.out.println("Sending SMS: " + formattedMessage);
    }
}
/**
 * The  EmailSender  class is a concrete implementation of the  MessageSender  abstract class.
 * It is responsible for sending email messages using a specified message formatter.
 */
public class EmailSender extends MessageSender {
    public EmailSender(MessageFormatter formatter) {
        super(formatter);
    }

    @Override
    public void sendMessage(String message) {
        String formattedMessage = formatter.formatMessage(message);

        // Send email with the formatted message
        System.out.println("Sending email: " + formattedMessage);
    }
}
```

 In this example, we have the *MessageSender* abstraction representing the functionality of sending messages. The *MessageFormatter* interface acts as the implementor, defining the low-level formatting functionality. The concrete implementor classes, *PlainTextFormatter* and *HtmlFormatter*, provide specific implementations of the formatting logic. The *EmailSender* and *SmsSender* classes act as refined abstractions, extending the *MessageSender* class and implementing the *sendMessage()* method using the appropriate formatting strategy. The *MessagingSystem* class demonstrates the usage of the Bridge pattern by creating instances of different senders with their respective formatters and invoking the *sendMessage()* method. The messages are formatted and sent according to the chosen formatting strategy. Bridge pattern enables the separation of the message sending functionality (*MessageSender*) from the formatting logic (*MessageFormatter*), allowing them to vary independently and providing flexibility and maintainability in the messaging system.

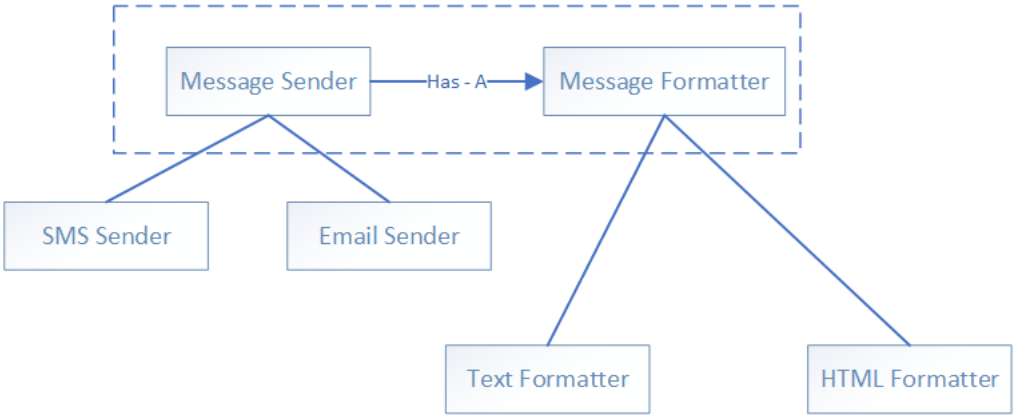

Figure 5-0-2

🎯 Bridge pattern decouples the abstraction (high-level functionality) from the implementation (low-level details). It achieves this by using composition, which establishes a **"Has-A**" relationship between the abstraction and the implementation. It allows both to vary independently, reducing the dependencies between them. This promotes flexible design and makes it easier to modify or extend the system without impacting other parts of the codebase. The Bridge pattern aligns with the **Single Responsibility Principle (SRP)** by separating the responsibilities of the abstraction and implementation into separate classes. Each class has a clear and focused responsibility, making the codebase more modular and easier to understand. It also facilitates the adherence to other **SOLID** principles such as **Open-Closed Principle (OCP)** and **Dependency Inversion Principle (DIP)**.

🚨 Designing the abstraction (abstraction interface or abstract class) requires careful consideration. It should focus on the essential high-level functionality and not become overly specific or too granular. Defining a proper abstraction is crucial to ensure a clear separation between the abstraction and implementation, facilitating future modifications and extensions.

📣 While the Bridge pattern and Object Adapter pattern may share some structural similarities, they have distinct intents and purposes. The Bridge pattern focuses on decoupling the abstraction (interface) from its implementation, allowing them to vary independently. It aims to provide a bridge between the abstraction and implementation, enabling them to evolve and change separately. The Bridge pattern promotes flexibility, extensibility, and the ability to switch implementations at runtime.

Adapter pattern is designed to adapt the interface of an existing object to make it compatible with another interface. It allows objects with incompatible interfaces to work together by wrapping the existing object with an adapter class. The adapter class translates the requests from the target interface into calls to the adapted object's interface.

1.3. Composite Pattern

💡 ***Composite Pattern*** *is used to arrange objects in tree like structure to represent part-whole hierarchies, where the whole can be composed of sub-parts, which in turn can be composed of further sub-parts, forming a recursive structure.*

We often deal with hierarchical systems in our daily life. If we are going to Bank or Post office, they have different hierarchical system in place. A sports team will have its own hierarchy, even computers have directories and subdirectories. Graphic User Interfaces (GUIs) have components which can be structured in hierarchy. Hierarchies can be simple or complex and are often represented using tree like structure in diagrams. How do we represent such a system in Software Programs?

Consider a scenario where we are developing an employee management system for a large organization. The system needs to handle hierarchical relationships among employees, such as managers and their subordinates. Each employee can have their own set of properties and methods specific to their role.

Initially, we might design separate classes for each role, such as Manager, Supervisor, and Employee, with their respective attributes and behaviours. However, as the organization grows and the hierarchy becomes more complex, we realize that the existing design is not scalable and leads to tightly coupled code. Say, if we wanted to perform an operation on a group of employees, we would need to write separate code to handle each level of the hierarchy, leading to tightly coupled and duplicated logic.

 Hence, we leverage **composite pattern** to represent the hierarchical structure of employees as a tree-like structure, where can perform operations on individual employees or groups of employees without the need for separate code paths.

 Let us look at the code snippet below to understand better.

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

// Component interface
interface EmployeeComponent {
    void displayInformation();
}

// Leaf class representing an individual employee
class Employee implements EmployeeComponent {
    private String name;
    private String role;

    public Employee(String name, String role) {
        this.name = name;
        this.role = role;
    }

    public void displayInformation() {
        System.out.println("Employee: " + name + ", Role: " + role);
    }
}

// Composite class representing a supervisor and their subordinates (employees)
class Supervisor implements EmployeeComponent {
    private String name;
    private List<EmployeeComponent> subordinates;

    public Supervisor(String name) {
        this.name = name;
        this.subordinates = new ArrayList<>();
    }

    public void addSubordinate(EmployeeComponent subordinate) {
        subordinates.add(subordinate);
    }

    public void removeSubordinate(EmployeeComponent subordinate) {
        subordinates.remove(subordinate);
    }

    public void displayInformation() {
        System.out.println("Supervisor: " + name);
        System.out.println("Subordinates:");
```

```
        for (EmployeeComponent subordinate : subordinates) {
            subordinate.displayInformation();
        }
    }
}

// Composite class representing a manager and their subordinates (supervisors)
class Manager implements EmployeeComponent {
    private String name;
    private List<EmployeeComponent> subordinates;

    public Manager(String name) {
        this.name = name;
        this.subordinates = new ArrayList<>();
    }

    public void addSubordinate(EmployeeComponent subordinate) {
        subordinates.add(subordinate);
    }

    public void removeSubordinate(EmployeeComponent subordinate) {
        subordinates.remove(subordinate);
    }

    public void displayInformation() {
        System.out.println("Manager: " + name);
        System.out.println("Subordinates:");
        for (EmployeeComponent subordinate : subordinates) {
            subordinate.displayInformation();
        }
    }
}

// Client code
public class EmployeeManagementSystem {
    public static void main(String[] args) {
        // Creating individual employees
        Employee john = new Employee("John Smith", "Developer");
        Employee anna = new Employee("Anna Johnson", "QA Engineer");

        // Creating a supervisor and adding employees
        Supervisor supervisor = new Supervisor("Michael Scott");
        supervisor.addSubordinate(john);
        supervisor.addSubordinate(anna);

        // Creating a manager and adding a supervisor
        Manager manager = new Manager("David Wallace");
        manager.addSubordinate(supervisor);

        // Displaying information
        manager.displayInformation();
    }
}
```

 If we look at the code above closely, we will observe that *displayInformation()* method, when called on a manager object, would recursively call the same method on all its subordinate in a hierarchical order. Also, this tree structure allows flexibility for a manager to add either employee or

Supervisor as its child components (see Figure 5-3).

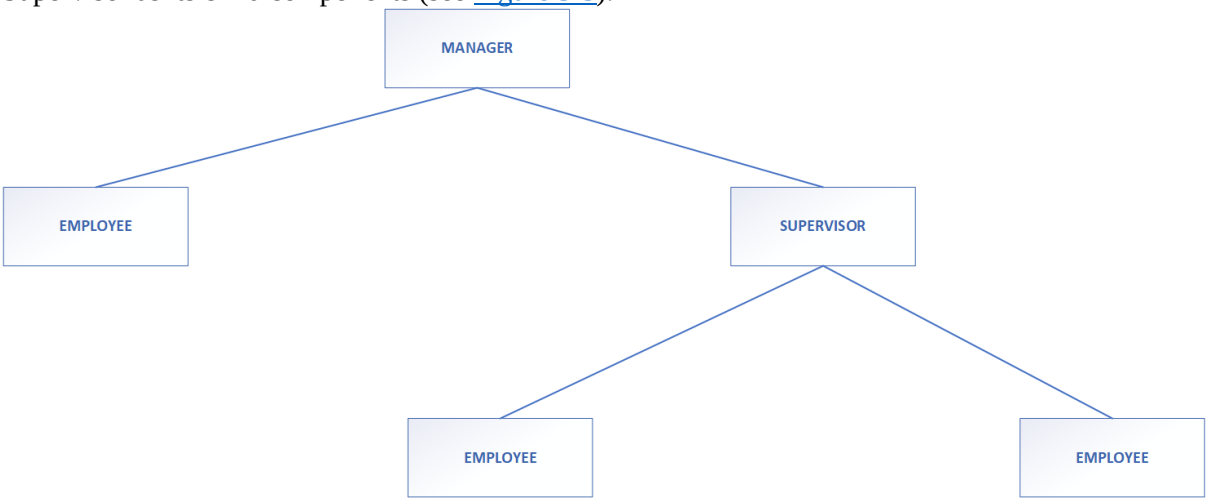

Figure 5-0-3

Composite Pattern has three key elements:

1. **Component:** The Component represents the common interface or base class for all elements in the hierarchy, whether they are individual objects or compositions of objects. It defines the basic operations that can be performed on the elements, such as adding, removing, or accessing child elements. In above example, *EmployeeComponent* is component element.
2. **Leaf:** The Leaf is the implementation of the Component interface for individual objects that do not have any child elements. Leaf objects represent the building blocks or basic elements of the hierarchy. In above example, *Employee* is a Leaf element.
3. **Composite:** The Composite represents the implementation of the Component interface for composite objects. Composite objects are containers that can hold other components, including both leaf objects and other composite objects. They delegate operations to their child components, recursively traversing the tree structure. In above example, *Manager* and *Supervisor* are composite elements.

Composite Pattern enables the creation of complex structures that can be treated uniformly, simplifying the code, and making it more flexible and scalable. It promotes code reusability by allowing the use of the same operations and interfaces for individual objects and compositions. Additionally, the Composite pattern supports recursive traversal and operations on the entire structure, enabling efficient manipulation and processing of complex hierarchies. It also allows for dynamic additions or removals of objects from the structure, providing flexibility in managing and modifying the composition.

1.4. Decorator Pattern

Decorator pattern *is a structural design pattern that allows behavior to be added to an individual object dynamically and offer a flexible alternative to subclassing when it comes to extending the functionality of an object.*

Extension of functionality is a common business use case in real world. Successful products keep extending their functionalities to be ahead in the game. Small businesses keep extending their functionality to survive and thrive. And when we integrate their requirements in software development, we encounter frequent request of functionality addition or deletion. If something is not working, it should be removed quickly and if there is something new, which when added upon existing functionality, boosts sales, it should be integrated in a jiffy.


And this is where Decorator Patterns come in handy. They help in extending functionality of an object without altering or affecting the existing behavior. They also help in giving an option for mix and match for choosing functionality on run time.

Okay, enough of theory and let us consider a potential business scenario and then see its implementation use case.

Imagine a coffee shop wants to implement a custom coffee ordering system to cater to the unique preferences of its customers. The system should allow customers to create personalized coffee orders by selecting various options such as type of coffee, additional ingredients, and toppings.

So, we have a base entity *Coffee* which needs to be customized and as per customization its features such as cost and description would change. It makes sense, right? A simple coffee would cost differently than a milk coffee or a cream whipped coffee.


If we adhere to traditional approach to address this requirement (see Figure 5-4), we may end up creating new subclasses, say *MilkCoffee* and *WhippedCreamCoffee* in our system. And when a customer asks for Milk Coffee with whipped cream, then what do we do? Create another subclass? And when our system has many such subclasses, then imagine the chaos of subclasses to be maintained. Something is wrong with this approach and we will fix it right after a coffee break!

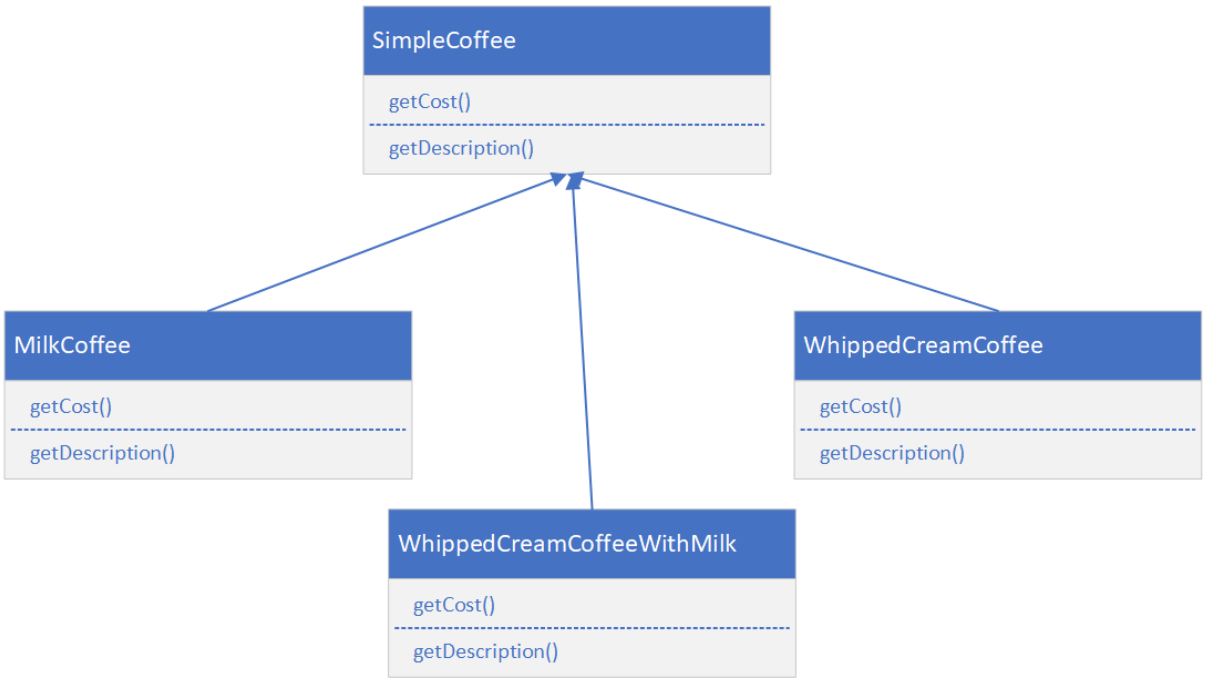

Figure 5-0-4


To allow dynamic addition of functionalities at run time, **Decorator Pattern** is used. By using decorators, we can wrap the original object with one or more decorators, each providing a specific extension or modification to the object's behavior. This approach avoids the need for an excessive number of subclasses and reduces the complexity of the class hierarchy.

Furthermore, decorators follow the principle of **composition over inheritance**, as they allow objects to be composed of multiple decorators, each responsible for a specific aspect of functionality. This composition-based approach promotes code reusability, modularity, and maintainability.

We will see a code snippet which leverages Decorator Pattern to give us flexibility to extend functionality of our system dynamically and it also does not alter existing functionality. Cool!

```
/**
 * The Coffee interface represents a type of coffee.
 * It provides methods to get the cost and description of the coffee.
 */
public interface Coffee {
     /**
     * Returns the cost of the coffee.
     *
     * @return the cost of the coffee as a double value
     */
    double getCost();
     /**
     * Returns the description of the coffee.
     *
     * @return the description of the coffee as a String
     */
    String getDescription();
}
/**
 * The SimpleCoffee class represents a simple type of coffee.
 * It implements the Coffee interface and provides the implementation for its methods.
 */
public class SimpleCoffee implements Coffee {
    @Override
    public double getCost() {
        return 1.0;
    }

    @Override
    public String getDescription() {
        return "Simple Coffee";
    }
}
/**
 * The CoffeeDecorator class is an abstract class that represents a decorator for a Coffee
object.
 * It implements the Coffee interface and provides a common implementation for its methods.
 * Subclasses of CoffeeDecorator can add additional behavior to the decorated coffee object.
 */
public abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

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

    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }
}
/**
 * The MilkDecorator class represents a decorator that adds milk to a Coffee object.
 * It extends the CoffeeDecorator class and provides additional behavior for the decorated
coffee.
 */
```

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

    @Override
    public double getCost() {
        return super.getCost() + 0.5;
    }

    @Override
    public String getDescription() {
        return super.getDescription() + ", Milk";
    }
}
/**
 * The WhippedCreamDecorator class represents a decorator that adds WhippedCream to a Coffee
object.
 * It extends the CoffeeDecorator class and provides additional behavior for the decorated
coffee.
 */
public class WhippedCreamDecorator extends CoffeeDecorator {
    public WhippedCreamDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    @Override
    public double getCost() {
        return super.getCost() + 0.75;
    }

    @Override
    public String getDescription() {
        return super.getDescription() + ", Whipped Cream";
    }
}
/**
 * The Client class demonstrates the usage of the Coffee decorators.
 * It creates instances of different coffee objects and prints their cost and description.
 */
public class Client {
    public static void main(String[] args) {
        Coffee simpleCoffee = new SimpleCoffee();
        System.out.println("Cost: " + simpleCoffee.getCost() + ", Description: " +
simpleCoffee.getDescription());

        Coffee milkCoffee = new MilkDecorator(simpleCoffee);
        System.out.println("Cost: " + milkCoffee.getCost() + ", Description: " +
milkCoffee.getDescription());

        Coffee whippedCreamCoffee = new WhippedCreamDecorator(milkCoffee);
        System.out.println(
                "Cost: " + whippedCreamCoffee.getCost() + ", Description: " +
whippedCreamCoffee.getDescription());

    }
}
```

 In the code example given above, the Decorator pattern is implemented using the *CoffeeDecorator* abstract class and its concrete subclasses such as *MilkDecorator* and *WhippedCreamDecorator*.

The *CoffeeDecorator* class acts as a wrapper for the Coffee object and implements the Coffee interface. It holds a reference to the decorated coffee object (*decoratedCoffee*) and delegates calls to

the underlying object for methods such as *getCost()* and *getDescription()*. This allows the decorator to modify or extend the behavior of the decorated object.

Each concrete decorator class, such as *MilkDecorator* and *WhippedCreamDecorator*, extends the *CoffeeDecorator* class. These decorators add specific functionalities by overriding the *getCost()* and *getDescription()* methods and providing their own behavior. They also make use of the **super** keyword to call the corresponding methods of the decorated object.

 **Decorators** allow for the dynamic addition of new functionalities to an object at runtime. We can wrap the original *Coffee* object with one or more decorators, adding or removing functionality as needed. This dynamic composition provides flexibility without modifying the original object or affecting other objects using the same interface. This enables fine-grained control over the behavior and allows for flexible customization. Adhering to SRP, each decorator class has a single responsibility, adding or modifying a specific behavior. This promotes code modularity and separation of concerns, making the system easier to understand, modify, and maintain.

 Decorators wrap the original object, adding or modifying its behavior, but they do not change the underlying object's identity. The decorated component and the decorator are distinct objects, even though the decorator delegates most of its functionality to the underlying component.

 When using the Decorator pattern extensively, it is common for systems to be composed of many small objects that appear similar. The differences between these objects lie primarily in their interconnections and not in their class or variable values.

This composition of numerous interconnected objects can make the system complex and potentially challenging to understand, learn, and debug, especially for developers who are new to the codebase. The similarities between the objects, combined with the dynamic nature of decorators, can introduce additional complexity and make it harder to trace and debug issues.

1.5. Facade Pattern

 ***Façade Pattern*** *provides a simplified interface to a library, a framework, or complex subsystem of classes, making it easier to use and understand. It acts as a high-level interface that hides the complexities of the underlying system and provides a unified and simplified interface for clients to interact with.*

Imagine using a coffee vending machine after a tiring day to treat yourself to a cup of coffee. Instead of having to go through the individual steps of pouring milk, adding coffee, and selecting flavors, the vending machine provides a simplified interface through a single button on its GUI. By pressing the button for a specific flavored coffee, the vending machine automatically handles the process of pouring milk, adding coffee, and incorporating the chosen flavor. This simplified interface acts as a ***Facade****,* ***abstracting away the complexities of the internal workings*** *of the machine and providing a convenient and streamlined experience for the user.*

Let us take another example in terms of software development.

Imagine we have a powerful and complex sound editing software that offers a vast array of features for professional audio production. However, we are working on a simple mobile app that only needs to perform basic audio trimming and volume adjustment for user-generated content.

 In this scenario, integrating the entire sound editing software directly into our mobile app would be overkill and could lead to unnecessary complexity. Users of mobile app would be exposed with lot many features which they have no use of and may end getting confused. We want to keep our interface clean and simple to use and focus on the main goal which is basic audio trimming and volume adjustment.

 **Façade Pattern** can be use here to shift focus of the app back to the specific functionalities it needs, making it more lightweight, easier to maintain, and user-friendly. It shields the app from the complexities of the extensive library, acting as a convenient interface for the limited scope of audio editing needed for our app.

 Let us look at the sample code snippet to understand better.

```
/**
 * This class represents a sound editing software that provides various functionalities
 * for manipulating audio files.
 */
public class SoundEditingSoftware {
     /**
     * Opens the specified audio file.
     *
     * @param filePath the path of the audio file to be opened
     */
    public void openFile(String filePath) {
        System.out.println("Opening file: " + filePath);
        // Perform complex operations to open the audio file
    }
     /**
     * Trims the audio file from the specified start time to the specified end time.
     *
     * @param startTime the start time in seconds
     * @param endTime   the end time in seconds
     */
    public void trimAudio(double startTime, double endTime) {
        System.out.println("Trimming audio from " + startTime + "s to " + endTime + "s");
        // Perform complex operations to trim the audio
    }
     /**
     * Adjusts the volume of the audio by the specified percentage.
     *
     * @param percentage the percentage by which to adjust the volume
     */
    public void adjustVolume(int percentage) {
        System.out.println("Adjusting volume by " + percentage + "%");
        // Perform complex operations to adjust the audio volume
    }
     // Other complex functionalities of the sound editing software
}

/**
 * This class represents a facade for a sound editing software, providing simplified methods
 * for trimming audio files and adjusting their volume.
 */
public class SoundEditingFacade {
    private SoundEditingSoftware software;
     /**
     * Constructs a new SoundEditingFacade object.
     * Initializes the underlying SoundEditingSoftware instance.
     */
    public SoundEditingFacade() {
        software = new SoundEditingSoftware();
```

```
    }
     /**
     * Trims the audio file located at the specified filePath from the given startTime to the
endTime.
     *
     * @param filePath   the path of the audio file to be trimmed
     * @param startTime  the start time in seconds
     * @param endTime    the end time in seconds
     */
    public void trimAudio(String filePath, double startTime, double endTime) {
        software.openFile(filePath);
        software.trimAudio(startTime, endTime);
    }
     /**
     * Adjusts the volume of the audio file located at the specified filePath by the given
percentage.
     *
     * @param filePath    the path of the audio file to have its volume adjusted
     * @param percentage  the percentage by which to adjust the volume
     */
    public void adjustVolume(String filePath, int percentage) {
        software.openFile(filePath);
        software.adjustVolume(percentage);
    }
}
/**
 * This class represents a mobile app that demonstrates the usage of the SoundEditingFacade
class
 * to trim audio files and adjust their volume.
 */
public class MobileApp {
    /**
     * The main method that is executed when running the MobileApp.
     * It creates an instance of SoundEditingFacade and demonstrates the usage of its methods.
     *
     * @param args the command line arguments (not used in this example)
     */
    public static void main(String[] args) {
        SoundEditingFacade soundEditor = new SoundEditingFacade();
        String filePath = "audio_file.mp3";
        double startTime = 10.5;
        double endTime = 30.0;
        soundEditor.trimAudio(filePath, startTime, endTime);
        int volumePercentage = 50;
        soundEditor.adjustVolume(filePath, volumePercentage);
    }
}
```

 In this code example, we have a *SoundEditingSoftware* class representing a complex sound editing library with various functionalities. The *SoundEditingFacade* acts as a simplified interface to the library, providing methods like *trimAudio()* and *adjustVolume()* that internally delegate the operations to the *SoundEditingSoftware*.

The *MobileApp* class demonstrates the usage of the Facade pattern in the mobile app. It creates an instance of *SoundEditingFacade* and uses its methods to perform audio trimming and volume adjustment. The app can utilize the necessary sound editing features without being exposed to the complexities of the underlying sound editing software.

 **Facade pattern** is a useful design pattern in various scenarios where you need to simplify and provide a unified interface to a complex subsystem. By encapsulating the complexities behind a Facade, clients can interact with the system using a straightforward and intuitive API, without needing to understand the intricate details of the underlying components. It improves code maintainability by

providing a single-entry point for accessing the subsystem functionality. Changes or updates to the subsystem can be localized within the Facade implementation, without requiring modifications to the client code.

 When used in conjunction with the Facade pattern, the Abstract Factory pattern helps to provide an interface for creating subsystem objects in a subsystem-independent manner. The Facade acts as a simplified interface to the complex subsystem, while the Abstract Factory abstracts the creation of subsystem objects behind an interface. This combination allows clients to interact with the subsystem through the Facade without being concerned about the specific implementation details or platform-specific classes. And in cases, when only one Façade object is required, we can create Façade objects as Singletons.

1.6. Flyweight Pattern

 ***Flyweight Pattern*** *aims to optimize memory usage by sharing common data between multiple objects.*

Consider a game that requires rendering a large number of trees on a game map. The trees can be of different types, such as pine trees, oak trees, and birch trees. Each tree type has a specific appearance, but they share common properties like size and position on the map. hence, each individual tree object would store its complete state, including properties such as size, position, and appearance. Initially we go ahead and create objects as and when required in the game and it works pretty well on our computer or phone. However, once we ship it for other users to explore this game and play, we get reports of system crashes. What went wrong?

 As the game map expands and more trees are added, the memory usage of the system would skyrocket. Each tree instance would consume a significant amount of memory, resulting in unnecessary duplication of data. The performance of the system would suffer due to the increased memory overhead and the need to create and manage many individual tree objects. Rendering each tree would require redundant computations and memory accesses, leading to slower frame rates and a less responsive gameplay experience. If we extend the system with new tree types, system will become more complex and error-prone, as each new type would require the creation of additional individual tree objects.

 Flyweight pattern can be used to optimize memory usage and improve performance. The key idea behind the Flyweight pattern is to split an object into two parts: **intrinsic state** and **extrinsic state**. The intrinsic state represents the shared data that can be shared among multiple objects, while the extrinsic state represents the unique data that varies for each object.

By separating the intrinsic and extrinsic state, the Flyweight pattern allows multiple objects to share the same intrinsic state, reducing memory consumption. The intrinsic state is typically stored in a flyweight factory or flyweight pool, which manages the creation and sharing of flyweight objects. The client objects use the flyweight objects by providing or manipulating the extrinsic (unique) state. Clients typically interact with the flyweight factory to obtain the required flyweight objects.

 Let us see a code snippet to understand it further.

```
import java.awt.Color;
import java.util.HashMap;
import java.util.Map;
 /**
 * Flyweight interface representing a tree.
```

```java
 */
interface Tree {
    /**
     * Renders the tree at the specified coordinates.
     *
     * @param x The x-coordinate of the tree.
     * @param y The y-coordinate of the tree.
     */
    void render(int x, int y);
}
 /**
 * Concrete flyweight class representing a specific type of tree (PineTree).
 */
class PineTree implements Tree {
    private final Color color;
     /**
     * Constructs a PineTree object.
     * Assume pine trees have a specific color (GREEN).
     */
    public PineTree() {
        color = Color.GREEN;
    }
     /**
     * Renders the pine tree at the specified coordinates with its color.
     *
     * @param x The x-coordinate of the tree.
     * @param y The y-coordinate of the tree.
     */
    @Override
    public void render(int x, int y) {
        System.out.println("Rendering a pine tree at (" + x + ", " + y + ") with color " +
color);
    }
}
 /**
 * Flyweight factory class responsible for creating and managing flyweight objects.
 */
class TreeFactory {
    private final Map<String, Tree> treeCache;
     /**
     * Constructs a TreeFactory object.
     * Initializes the tree cache.
     */
    public TreeFactory() {
        treeCache = new HashMap<>();
    }
     /**
     * Retrieves a tree object from the cache based on the specified type.
     * If the tree object does not exist in the cache, a new one is created and added to the
cache.
     *
     * @param type The type of tree to retrieve.
     * @return The tree object.
     */
    public Tree getTree(String type) {
        Tree tree = treeCache.get(type);
         if (tree == null) {
            // Create a new tree instance and add it to the cache
            if (type.equals("pine")) {
                tree = new PineTree();
            }
            // Additional tree types can be added here
             treeCache.put(type, tree);
        }
         return tree;
    }
```

```
}
 /**
 * Client class that uses flyweight objects.
 */
class Game {
    private final TreeFactory treeFactory;
     /**
     * Constructs a Game object with a TreeFactory.
     *
     * @param treeFactory The TreeFactory object to use for creating trees.
     */
    public Game(TreeFactory treeFactory) {
        this.treeFactory = treeFactory;
    }
     /**
     * Renders a tree of the specified type at the specified coordinates using the
TreeFactory.
     *
     * @param type The type of tree to render.
     * @param x    The x-coordinate of the tree.
     * @param y    The y-coordinate of the tree.
     */
    public void renderTree(String type, int x, int y) {
        Tree tree = treeFactory.getTree(type);
        tree.render(x, y);
    }
}
/**
 * An example usage of the Flyweight pattern.
 */
public class FlyweightExample {
     /**
     * The main entry point of the program.
     *
     * @param args The command line arguments.
     */
    public static void main(String[] args) {
        TreeFactory treeFactory = new TreeFactory();
        Game game = new Game(treeFactory);
         // Render multiple pine trees at different locations
        game.renderTree("pine", 10, 20);
        game.renderTree("pine", 30, 40);
        game.renderTree("pine", 50, 60);
         // Additional tree types can be rendered here
    }
}
```

 In above code example, we have a *Tree* interface representing the flyweight objects, and a concrete flyweight class *PineTree* that implements the interface. The *TreeFactory* class acts as the flyweight factory, responsible for creating and managing the flyweight objects. The *Game* class is the client that uses the flyweight objects to render trees.

When the *renderTree* method is called in the Game class, it retrieves the flyweight object from the *TreeFactory* based on the tree type. If the object already exists in the cache (see usage of HashMap), it is retrieved and used. If not, a new flyweight object is created, added to the cache, and then used. The flyweight objects are then rendered with their respective locations.

 By utilizing the Flyweight pattern, the common state (in this case, the color of the tree) is shared among multiple tree objects, resulting in memory efficiency. The flyweight objects are created and cached by the factory, allowing them to be reused when needed. It also reduced garbage collection

overhead. It promotes a clear separation of concerns by separating the intrinsic and extrinsic state. This makes the codebase more maintainable and easier to understand.

 Since, we are always returning the used instance of a flyweight object, we might confuse it with Singleton pattern. However, the Flyweight pattern and the Singleton pattern are distinct design patterns with different purposes and behaviours. The Flyweight pattern is focused on optimizing memory usage by sharing common data among multiple objects. On the other hand, the Singleton pattern is concerned with ensuring that only a single instance of a class exists throughout the application's lifetime. While both patterns can have a single instance, their purposes and usage differ.

The combination of the Flyweight and Composite patterns can be seen in scenarios where the Composite pattern is used to represent a logically hierarchical structure, and the Flyweight pattern is applied to share leaf nodes within that structure. This combination helps to minimize memory usage by reusing shared leaf nodes across different composite nodes.

By utilizing the Flyweight pattern for the leaf nodes, we can ensure that common leaf nodes are shared among multiple composite nodes, reducing memory consumption. This is particularly beneficial in cases where the leaf nodes have significant shared state or when the number of leaf nodes is large. It optimizes memory usage by reusing common leaf nodes and provides a unified interface for working with both individual leaf nodes and composite nodes.

1.7. Proxy Pattern

 ***Proxy Pattern****, as the name suggests, provides with a substitute or placeholder for another object. It allows us to control the access to the target object, providing additional functionalities or behaviors without changing the original object's interface.*

 But why would we proxy or substitute another object?

Answer is quite intuitive. If we could afford to carry an original object all the time, then we would have carried it in first place rather than proxying. But the reason we are proxying it suggests that it is costly to carry the burden of the object all the time. Still confused?

> *Say in a remote communication scenario, a client needs to interact with a remote object over a network. In a distributed environment, where different components are located on different servers, we take example of one such server, File Server that stores and manages files. The clients need to interact with the file server to perform file operations like reading, writing, and deleting files. It consumes good amount of system resources. We need instance of file server from time to time but not all the time. How do we manage to do that?*

 We can do **lazy initialization**, an approach where an object is created or initialized only when it is actually needed. However, this initialization code would need to be implemented in all the clients of File Server, which would result in code duplication.

 Proxy pattern help us by letting us implement the same interface as the original service object. However, the proxy does not create a new real service object upon receiving a request from a client. Instead, the proxy intercepts the client's request and can perform additional operations before or after delegating the actual work to the real service object.

The Proxy pattern involves the following components:

1. **Subject**: This is the interface or abstract class that defines the common interface between the Proxy and the target object. It specifies the methods that the Proxy should implement to mimic the behavior of the target object.
2. **Real Subject**: This is the actual object that the Proxy represents. It implements the Subject interface and contains the core functionality that the Proxy wraps.
3. **Proxy**: This is the surrogate object that acts as a substitute for the Real Subject. It implements the Subject interface and maintains a reference to the Real Subject. The Proxy intercepts client requests and performs additional actions if required, and then delegates the request to the Real Subject.

 Let us check a code snippet to understand this implementation.

```
/**
 * This interface represents a file server that allows reading, writing, and deleting files.
 */
public interface FileServer {
     /**
     * Reads the contents of a file with the specified filename.
     *
     * @param filename the name of the file to be read
     */
    void readFile(String filename);
     /**
     * Writes the given data to a file with the specified filename.
     *
     * @param filename the name of the file to be written
     * @param data     the data to be written to the file
     */
    void writeFile(String filename, byte[] data);
     /**
     * Deletes the file with the specified filename.
     *
     * @param filename the name of the file to be deleted
     */
    void deleteFile(String filename);
}
/**
 * This class represents a remote file server that implements the {@link FileServer}
interface.
 * It provides methods to read, write, and delete files on the remote server.
 */
public class RemoteFileServer implements FileServer {
    /**
     * Reads the contents of a file from the remote file server.
     *
     * @param filename the name of the file to be read
     */
    @Override
    public void readFile(String filename) {
        // Connect to the remote file server and perform the read operation
        System.out.println("Reading file: " + filename);
    }
     /**
     * Writes the given data to a file on the remote file server.
     *
     * @param filename the name of the file to be written
     * @param data     the data to be written to the file
     */
    @Override
    public void writeFile(String filename, byte[] data) {
        // Connect to the remote file server and perform the write operation
        System.out.println("Writing file: " + filename);
```

```
    }
     /**
     * Deletes a file from the remote file server.
     *
     * @param filename the name of the file to be deleted
     */
    @Override
    public void deleteFile(String filename) {
        // Connect to the remote file server and perform the delete operation
        System.out.println("Deleting file: " + filename);
    }
}
/**
 * This class represents a proxy for the {@link FileServer} interface.
 * It provides a convenient way to interact with a remote file server located at the specified
server address.
 * The proxy lazily initializes the remote file server and forwards read, write, and delete
operations to it.
 */
public class FileServerProxy implements FileServer {
    private final String serverAddress;
    private RemoteFileServer remoteFileServer;
     /**
     * Constructs a new FileServerProxy with the specified server address.
     *
     * @param serverAddress the address of the remote file server
     */
    public FileServerProxy(String serverAddress) {
        this.serverAddress = serverAddress;
    }
     /**
     * Reads the contents of a file from the remote file server.
     *
     * @param filename the name of the file to be read
     */
    @Override
    public void readFile(String filename) {
        // Lazy initialization of the remote file server
        if (remoteFileServer == null) {
            remoteFileServer = new RemoteFileServer();
        }
         // Forward the read operation to the remote file server
        remoteFileServer.readFile(filename);
    }
     /**
     * Writes the given data to a file on the remote file server.
     *
     * @param filename the name of the file to be written
     * @param data     the data to be written to the file
     */
    @Override
    public void writeFile(String filename, byte[] data) {
        // Lazy initialization of the remote file server
        if (remoteFileServer == null) {
            remoteFileServer = new RemoteFileServer();
        }
        // Forward the write operation to the remote file server
        remoteFileServer.writeFile(filename, data);
    }
     /**
     * Deletes a file from the remote file server.
     *
     * @param filename the name of the file to be deleted
     */
    @Override
    public void deleteFile(String filename) {
```

```
        // Lazy initialization of the remote file server
        if (remoteFileServer == null) {
            remoteFileServer = new RemoteFileServer();
        }
         // Forward the delete operation to the remote file server
        remoteFileServer.deleteFile(filename);
    }
}
/**
 * This class represents a client that interacts with a file server.
 * It demonstrates the usage of the {@link FileServer} interface and its proxy implementation.
 */
public class Client {
    /**
     * The entry point of the client application.
     *
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Create a file server proxy with the specified server address
        FileServer fileServer = new FileServerProxy("192.168.0.100");
         // Read a file from the file server
        fileServer.readFile("document.txt");
         // Write data to a file on the file server
        fileServer.writeFile("data.txt", new byte[] { 1, 2, 3 });
         // Delete a file from the file server
        fileServer.deleteFile("image.jpg");
    }
}
```

 In the code example above, *FileServer* is the **Subject** interface defining the file operations that clients can perform. *RemoteFileServer* is the **Real Subject**, which represents the actual file server running on a remote machine. *FileProxyServer* is the **Proxy**, which acts as a local representative for the remote file server. The Proxy lazily initializes the remote file server and forwards the file operations to the real server when requested by clients. The clients can interact with the file server through the Proxy, without being aware of the network communication and remote object details.

 By using the **Proxy pattern**, we can centralize the initialization logic and avoid code duplication unless caused by lazy initialization. The Proxy can handle the deferred initialization code, ensuring that it is executed only when the object is actually needed.

Key benefit of using Proxy is reducing the overhead of object creation or resource allocation until necessary. It adds access control mechanisms to the Real Subject, ensuring that certain operations are performed only by authorized clients.

It provides an additional layer of security by enforcing authentication, authorization, or other security checks. It captures method invocations and performs logging or auditing operations, such as recording method calls, collecting performance metrics, or logging error conditions.

It enhances the system's observability and allows for monitoring and analysis of the target object's behavior. Caching mechanism can be employed along side proxy to cache the results of expensive operations performed by the Real Subject, allowing subsequent requests with the same inputs to be served from the cache.

 Even though Decorator pattern and proxy Pattern may have similar implementation, their purpose is entirely different. Decorator is focused on adding additional responsibilities or behaviors to an object dynamically. Whereas, Proxy is primarily concerned with controlling access to an object.

Both patterns involve wrapping an object and providing additional functionality, but their intentions and primary focuses differ.

1.8. Summary

Structural design patterns are concerned with organizing classes and objects to form larger structures, focusing on relationships and interactions between them. These patterns help in achieving flexibility, modularity, and extensibility in software systems.

1. **Adapter Pattern:** Converts the interface of a class into another interface that clients expect. It allows incompatible classes to work together by acting as a bridge between them.
2. **Bridge Pattern:** Separates an abstraction from its implementation, allowing them to vary independently. It decouples the abstraction and implementation hierarchies, promoting flexibility and extensibility.
3. **Composite Pattern:** Composes objects into tree-like structures to represent part-whole hierarchies. It allows clients to treat individual objects and groups of objects uniformly.
4. **Decorator Pattern:** Dynamically adds responsibilities or behaviors to objects without modifying their underlying classes. It provides a flexible alternative to subclassing for extending functionality
5. **Facade Pattern:** Provides a simplified interface to a complex subsystem, acting as a high-level interface that makes the subsystem easier to use. It hides the complexities of the subsystem and provides a unified interface for clients.
6. **Flyweight Pattern:** Shares common state among multiple objects to minimize memory usage. It aims to optimize performance and memory consumption by reusing shared state instead of creating new instances.
7. **Proxy Pattern:** Controls access to an object by acting as a surrogate or placeholder. It provides a level of indirection, allowing additional operations to be performed before or after delegating to the real object.