Heaven's Light Is Our Guide

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Course Code: CSE 3206

Course Title: Software Engineering Sessional

Lab Report : Design Pattern(7,8,9)

Submitted by-

Name	Roll
Hasibul Hasan	2003127
Muhimenul Islam	2003128
KM Tanvir Ahmed	2003129

Submitted to-

Farjana Parvin
Lecturer,
Department of CSE, RUET

Bridge Structural Design Pattern:

Bridge Design Pattern is a structural design pattern that split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

The pattern involves two main components abstraction and implementation layer. Abstraction (also called interface) is a high-level control layer for some entity. This layer isn't supposed to do any real work on its own. It should delegate the work to the implementation layer (also called platform) which implements all the low-level functionality.

The Bridge Design Pattern allows for creating platform-independent classes and apps. In this design pattern client work with only the high-level abstractions. This pattern follows the Open/Closed Principle, enabling new abstractions and implementations to be added independently. The Bridge pattern attempts to solve this problem by switching from inheritance to composition.

Example with and without the Bridge Design Pattern

Let's consider an example where a **TV** can be controlled with different types of **remotes**. The TV is the **abstraction**, and the remote is the **implementation**. The TV can be controlled using different types of remote devices, such as **Standard Remote** and **Smart Remote**.

Without the Bridge Design Pattern:

Without the Bridge pattern, we would need to create separate subclasses for each combination of **TV** and **Remote** (e.g., LCDTVStandardRemote, LEDTVSmartRemote), which leads to a large class hierarchy.

```
class LCDTV {
    public void turnOn() {
        System.out.println("LCD TV is now ON.");
    }

    public void turnOff() {
        System.out.println("LCD TV is now OFF.");
    }
}

class LEDTV {
    public void turnOn() {
        System.out.println("LED TV is now ON.");
    }

    public void turnOff() {
        System.out.println("LED TV is now OFF.");
    }
}

// Remote class
```

```java
class StandardRemote {
    private LCDTV tv;

    public StandardRemote(LCDTV tv) {
        this.tv = tv;
    }

    public void powerButton() {
        tv.turnOn();
    }

    public void offButton() {
        tv.turnOff();
    }
}

class SmartRemote {
    private LCDTV tv;

    public SmartRemote(LCDTV tv) {
        this.tv = tv;
    }

    public void powerButton() {
        tv.turnOn();
    }

    public void offButton() {
        tv.turnOff();
    }
}

// Client code
public class WithoutBridgeExample {
    public static void main(String[] args) {
        LCDTV lcdTv = new LCDTV();
        StandardRemote standardRemote = new StandardRemote(lcdTv);

        standardRemote.powerButton();  // Turns on the LCD TV
        standardRemote.offButton();    // Turns off the LCD TV
    }
}
```

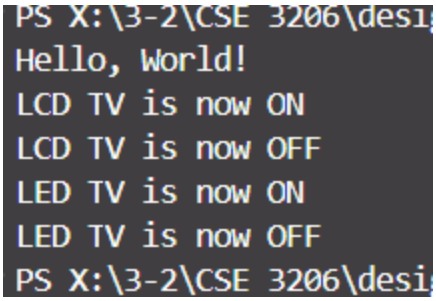
```
PS X:\3-2\CSE 3206\des
LCD TV is now ON.
LCD TV is now OFF.
```

With the Bridge Design Pattern:

With the Bridge pattern, it can design separating the abstraction (the TV) from the implementation (the remote), allowing the TV to change its implementation without needing to create a new class for each combination.

```
public class App {
    public static void main(String[] args) throws Exception {
        System.out.println("Hello, World!");

        TV lcdTv = new LCDTV(); // Create an LCD TV object
        Remote standardRemote = new StandardRemote(lcdTv); // Assign the TV to the
remote

        // Use the remote to control the TV
        standardRemote.powerButton();  // Turns on the LCD TV
        standardRemote.offButton();    // Turns off the LCD TV


        TV ledTv = new LEDTV(); // Create an LED TV object
        Remote smartRemote = new SmartRemote(ledTv); // Assign the TV to the remote
        smartRemote.powerButton();  // Turns on the LED TV
        smartRemote
            .offButton();    // Turns off the LED TV
    }
}

// With Bridge Design Pattern

// Implementor: Remote (Low-level functionality)
interface Remote {
    void powerButton();
    void offButton();
}

// Abstraction: TV (High-level functionality)
abstract class TV {
    public abstract void turnOn();
    public abstract void turnOff();
}

// Concrete Implementations of Remote
class StandardRemote implements Remote {
    private TV tv;

    // StandardRemote takes a TV object in its constructor
    public StandardRemote(TV tv) {
        this.tv = tv;
    }

    @Override
    public void powerButton() {
        tv.turnOn(); // Calls the turnOn method of the specific TV
    }
```

```
    @Override
    public void offButton() {
        tv.turnOff(); // Calls the turnOff method of the specific TV
    }
}

class SmartRemote implements Remote {
    private TV tv;

    // SmartRemote takes a TV object in its constructor
    public SmartRemote(TV tv) {
        this.tv = tv;
    }

    @Override
    public void powerButton() {
        tv.turnOn();
    }

    @Override
    public void offButton() {
        tv.turnOff();
    }
}

// Refined Abstraction: Specific TVs (LCD and LED)
class LCDTV extends TV {
    @Override
    public void turnOn() {
        System.out.println("LCD TV is now ON");
    }

    @Override
    public void turnOff() {
        System.out.println("LCD TV is now OFF");
    }
}

class LEDTV extends TV {
    @Override
    public void turnOn() {
        System.out.println("LED TV is now ON");
    }

    @Override
    public void turnOff() {
        System.out.println("LED TV is now OFF");
    }
}
```



The **Bridge Design Pattern** helps to decouple abstraction from implementation, making the system more flexible, easier to extend, and maintain. Without the Bridge pattern, the design would face rigid class hierarchies that are difficult to modify.

Composite Design Pattern

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


interface Graphic {
    void draw();  // Common method to be implemented by both Leaf and Composite
}

// Leaf class (represents simple objects, i.e., individual graphics)
class Circle implements Graphic {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}

class Rectangle implements Graphic {
    @Override
    public void draw() {
        System.out.println("Drawing a Rectangle");
    }
}

// Composite class (represents objects that can contain other objects, i.e.,
compositions)
class Drawing implements Graphic {
    private List<Graphic> graphics = new ArrayList<>();
```

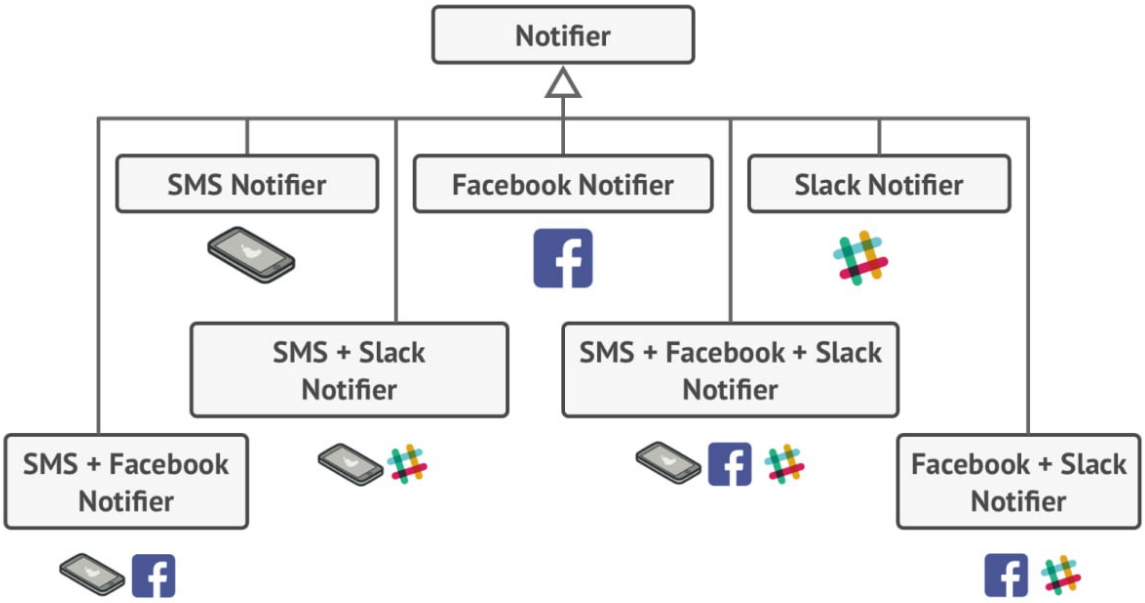
```
    // Add a graphic to the composite
    public void add(Graphic graphic) {
        graphics.add(graphic);
    }

    // Remove a graphic from the composite
    public void remove(Graphic graphic) {
        graphics.remove(graphic);
    }

    @Override
    public void draw() {
        for (Graphic graphic : graphics) {
            graphic.draw();  // Delegate the drawing to individual objects or nested
composites
        }
    }
}

// Client code to test the Composite Design Pattern
public class CompositePatternDemo {
    public static void main(String[] args) {
        // Create some leaf objects
        Graphic circle1 = new Circle();
        Graphic rectangle1 = new Rectangle();

        // Create a composite object (Drawing)
        Drawing drawing = new Drawing();
        drawing.add(circle1);
        drawing.add(rectangle1);

        // Create another composite object (Nested Drawing)
        Drawing nestedDrawing = new Drawing();
        nestedDrawing.add(new Circle());  // Nested circle

        // Add nested drawing to the main drawing
        drawing.add(nestedDrawing);

        // Draw everything
        System.out.println("Drawing everything:");
        drawing.draw();  // Will draw circles, rectangles, and nested drawings
    }
}
```

Decorator Design Pattern

Decorator is a structural design pattern that lets us attach new

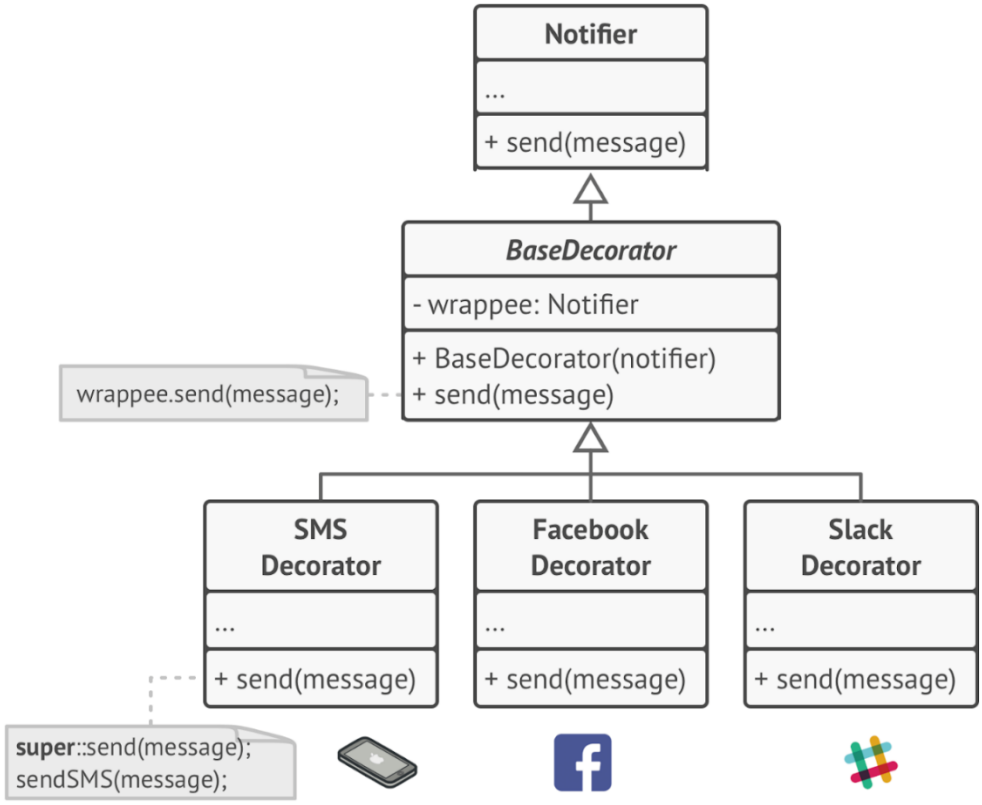
behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Problem: Extending Notification Capabilities Using the Decorator Pattern



Combinatorial explosion of subclasses.

UML diagram for notification alert system using Decoration Design Pattern



Various notification methods become decorators.

Code:

```
//Base notifier interface
interface Notifier {
    void send(String message);
}

// Email Notifier (Base Implementation)
class EmailNotifier implements Notifier {
```

```
    private String[] emails;

    public EmailNotifier(String[] emails) {
        this.emails = emails;
    }

    @Override
    public void send(String message) {
        for (String email : emails) {
            System.out.println("Sending Email to " + email + ": "
+ message);
        }
    }
}

// Base Decorator
abstract class NotifierDecorator implements Notifier {
    protected Notifier notifier;

    public NotifierDecorator(Notifier notifier) {
        this.notifier = notifier;
    }

    @Override
    public void send(String message) {
        notifier.send(message);
    }
}

// SMS Decorator
class SMSNotifier extends NotifierDecorator {
    private String[] phoneNumbers;

    public SMSNotifier(Notifier notifier, String[] phoneNumbers)
{
        super(notifier);
        this.phoneNumbers = phoneNumbers;
    }

    @Override
    public void send(String message) {
        super.send(message); // Call previous notifier
        for (String number : phoneNumbers) {
            System.out.println("Sending SMS to " + number + ": "
+ message);
        }
    }
```

```
}

// Facebook Decorator
class FacebookNotifier extends NotifierDecorator {
    private String[] fbAccounts;

    public FacebookNotifier(Notifier notifier, String[]
fbAccounts) {
        super(notifier);
        this.fbAccounts = fbAccounts;
    }

    @Override
    public void send(String message) {
        super.send(message); // Call previous notifier
        for (String account : fbAccounts) {
            System.out.println("Sending Facebook Message to " +
account + ": " + message);
        }
    }
}

// Slack Decorator
class SlackNotifier extends NotifierDecorator {
    private String[] channels;

    public SlackNotifier(Notifier notifier, String[] channels) {
        super(notifier);
        this.channels = channels;
    }

    @Override
    public void send(String message) {
        super.send(message); // Call previous notifier
        for (String channel : channels) {
            System.out.println("Sending Slack Message to " +
channel + ": " + message);
        }
    }
}

// Client Code
public class NotificationApp {
    public static void main(String[] args) {
        // Initial Email Notification
        Notifier emailNotifier = new EmailNotifier(new String[]
{"user1@example.com", "user2@example.com"});
```

```
        // Adding SMS Notification
        Notifier smsNotifier = new SMSNotifier(emailNotifier, new
String[] {"+1234567890", "+0987654321"});

        // Adding Facebook Notification
        Notifier fbNotifier = new FacebookNotifier(smsNotifier,
new String[] {"fb_user1", "fb_user2"});

        // Adding Slack Notification
        Notifier slackNotifier = new SlackNotifier(fbNotifier,
new String[] {"#general", "#alerts"});

        // Sending Notifications through all channels
        slackNotifier.send("Emergency Alert: Fire in the
building!");
    }
}
```