**WEEK 1**

**Additional Hands-On**

**SOLID PRINCIPLES AND DESIGN PATTERNS:**

**EX:5 Implementing the Decorator Pattern**

**The reason to use Decorator design pattern for adding notification channels dynamically is that:**

* It allows behaviour to be added to individual objects dynamically.
* It promotes the Open-Closed Principle by extending behaviour without modifying existing code.
* It enables flexible combinations of features at runtime.
* It keeps each class focused on a single responsibility.

**Component Interface and Decorator class (Definition and Implementation):**

public interface Notifier {

void send(String message);

}

public class EmailNotifier implements Notifier {

public void send(String message) {

System.out.println("Sending Email: " + message);

}

}

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message);

}

}

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

System.out.println("Sending SMS: " + message);

}

}

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

System.out.println("Sending Slack message: " + message);

}

}

**Testing the Decorator Implementation using Main class:**

public class Main {

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

notifier = new SMSNotifierDecorator(notifier);

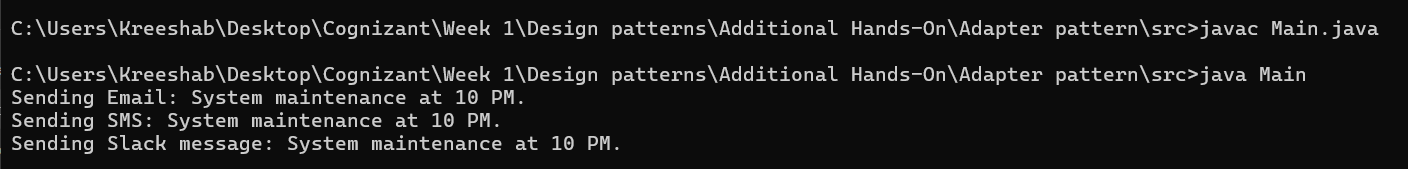
notifier = new SlackNotifierDecorator(notifier);

notifier.send("System maintenance at 10 PM.");

}

}

**TESTING OUTPUT:**

****