WEEK 1 : Design patterns and principles

Exercise 1: Implement Singleton Pattern

Code:

```
package week_1.Design_Patterns_Principles;
class Logger {
  private static Logger instance;
  private Logger() {
     System.out.println("Logger instance created.");
  }
  public static Logger getInstance() {
     if (instance == null) {
        instance = new Logger();
     }
     return instance;
  }
  public void log(String message) {
     System.out.println("Log: " + message);
  }
}
class SingletonPatternExample {
  public static void main(String[] args) {
     Logger logger1 = Logger.getInstance();
     logger1.log("First message");
     Logger logger2 = Logger.getInstance();
     logger2.log("Second message");
     System.out.println(logger1 == logger2);
  }
}
```

Output:

```
Problems  Javadoc  Declaration  Console ×
<terminated> SingletonPatternExample [Java Application] C:\Users
Logger instance created.
Log: First message
Log: Second message
true
```

Exercise 2: Implementing the Factory Method Pattern

Code:

```
package week_1.Design_Patterns_Principles;
interface Document {
void open();
}
class WordDocument implements Document {
```

```
public void open() {
   System.out.println("Opening Word Document...");
}
}
class PdfDocument implements Document {
public void open() {
   System.out.println("Opening PDF Document...");
}
}
class ExcelDocument implements Document {
public void open() {
   System.out.println("Opening Excel Document...");
}
}
abstract class DocumentFactory {
public abstract Document createDocument();
}
class WordDocumentFactory extends DocumentFactory {
public Document createDocument() {
   return new WordDocument();
}
}
class PdfDocumentFactory extends DocumentFactory {
public Document createDocument() {
   return new PdfDocument();
}
}
class ExcelDocumentFactory extends DocumentFactory {
public Document createDocument() {
   return new ExcelDocument();
}
}
public class FactoryMethodPatternExample {
public static void main(String[] args) {
   DocumentFactory factory = new WordDocumentFactory();
   Document doc = factory.createDocument();
   doc.open();
   factory = new PdfDocumentFactory();
   doc = factory.createDocument();
   doc.open();
   factory = new ExcelDocumentFactory();
   doc = factory.createDocument();
   doc.open();
} }
```

Output:

```
Problems  Javadoc  Declaration  Console ×
<terminated> FactoryMethodPatternExample [Java Applicatio
Opening Word Document...
Opening PDF Document...
Opening Excel Document...
```

⇒ Additional Important hands-on

Exercise 3: Implementing Builder Pattern

Code:

```
package week_1.Design_Patterns_Principles;
public class Computer {
  private String CPU;
  private String RAM;
  private String storage;
  private Computer(Builder builder) {
     this.CPU = builder.CPU;
     this.RAM = builder.RAM;
     this.storage = builder.storage;
  }
  public static class Builder {
     private String CPU;
     private String RAM;
     private String storage;
     public Builder setCPU(String cpu) {
        this.CPU = cpu;
        return this;
     }
     public Builder setRAM(String ram) {
        this.RAM = ram;
        return this;
     }
     public Builder setStorage(String storage) {
        this.storage = storage;
        return this;
     }
     public Computer build() {
        return new Computer(this);
     }
  }
  public String toString() {
     return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", Storage=" + storage + "]";
  }
  public static void main(String[] args) {
     Computer comp1 = new Computer.Builder()
           .setCPU("Intel i5")
           .setRAM("16GB")
           .setStorage("512GB SSD")
           .build();
     Computer comp2 = new Computer.Builder()
           .setCPU("AMD Ryzen 7")
           .setRAM("32GB")
           .setStorage("1TB SSD")
           .build();
     System.out.println(comp1);
     System.out.println(comp2);
```

```
  }
}
```

Output:

```
Problems  Javadoc  Declaration  Console ×
<terminated> Computer [Java Application] C:\Users\zerub\.p2\pool\plugins\org.eclipse.justj.op
Computer [CPU=Intel i5, RAM=16GB, Storage=512GB SSD]
Computer [CPU=AMD Ryzen 7, RAM=32GB, Storage=1TB SSD]
```

Exercise 4: Implementing the Adapter Pattern

Code:

```
package week_1.Design_Patterns_Principles;
interface PaymentProcessor {
void processPayment(double amount);
}
class PayPalGateway {
public void sendPayment(double amount) {
   System.out.println("Paid " + amount + " using PayPal.");
}
}
class StripeGateway {
public void makePayment(double amount) {
   System.out.println("Paid " + amount + " using Stripe.");
}
}
class PayPalAdapter implements PaymentProcessor {
private PayPalGateway paypal;
public PayPalAdapter(PayPalGateway paypal) {
   this.paypal = paypal;
}
public void processPayment(double amount) {
   paypal.sendPayment(amount);
}
}
class StripeAdapter implements PaymentProcessor {
private StripeGateway stripe;
public StripeAdapter(StripeGateway stripe) {
   this.stripe = stripe;
}
public void processPayment(double amount) {
   stripe.makePayment(amount);
}
}
public class AdapterPatternExample {
public static void main(String[] args) {
```

```
    PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalGateway());
    paypalProcessor.processPayment(100.0);
    PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());
    stripeProcessor.processPayment(200.0);
}
}
```

Output:

```
Problems  @ Javadoc  Declaration  Conso
<terminated> AdapterPatternExample [Java Applica
Paid 100.0 using PayPal.
Paid 200.0 using Stripe.
```

Exercise 5: Implementing the Decorator Pattern

Code:

```
package week_1.Design_Patterns_Principles;
interface Notifier {
void send(String message);
}
class EmailNotifier implements Notifier {
public void send(String message) {
    System.out.println("Email sent: " + message);
}
}
abstract class NotifierDecorator implements Notifier {
protected Notifier notifier;
public NotifierDecorator(Notifier notifier) {
    this.notifier = notifier;
}
public void send(String message) {
    notifier.send(message);
}
}
class SMSNotifierDecorator extends NotifierDecorator {
public SMSNotifierDecorator(Notifier notifier) {
    super(notifier);
}
public void send(String message) {
    super.send(message);
    System.out.println("SMS sent: " + message);
}
}
class SlackNotifierDecorator extends NotifierDecorator {
public SlackNotifierDecorator(Notifier notifier) {
    super(notifier);
```

```
}
public void send(String message) {
    super.send(message);
    System.out.println("Slack message sent: " + message);
}
}
public class DecoratorPatternExample {
public static void main(String[] args) {
    Notifier notifier = new EmailNotifier();
    Notifier smsNotifier = new SMSNotifierDecorator(notifier);
    Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);
    slackNotifier.send("System update completed.");
}
}
```

Output:

```
Problems  Javadoc  Declaration  Console ×
<terminated> DecoratorPatternExample [Java Application] C:\Users\zerub\.p2\
Email sent: System update completed.
SMS sent: System update completed.
Slack message sent: System update completed.
```

Exercise 6: Implementing Proxy Pattern

Code:

```
package week_1.Design_Patterns_Principles;
interface Image {
void display();
}
class RealImage implements Image {
private String filename;
public RealImage(String filename) {
    this.filename = filename;
    loadImageFromDisk();
}
private void loadImageFromDisk() {
    System.out.println("Loading " + filename);
}
public void display() {
    System.out.println("Displaying " + filename);
}
}
class ProxyImage implements Image {
private RealImage realImage;
private String filename;
public ProxyImage(String filename) {
    this.filename = filename;
```

```
}
public void display() {
    if (realImage == null) {
        realImage = new RealImage(filename);
    }
    realImage.display();
}
}
public class ProxyPatternExample {
public static void main(String[] args) {
    Image img1 = new ProxyImage("photo1.jpg");
    Image img2 = new ProxyImage("photo2.jpg");
    img1.display();
    img1.display();
    img2.display();
} }
```

Output:

```
Problems   Javadoc   Declaration
<terminated> ProxyPatternExample [Java A
Loading photo1.jpg
Displaying photo1.jpg
Displaying photo1.jpg
Loading photo2.jpg
Displaying photo2.jpg
```

Exercise 7: Implementing the Observer Pattern

Code:

```
package week_1.Design_Patterns_Principles;
import java.util.ArrayList;
import java.util.List;
interface Observer {
  void update(float price);
}
interface Stock {
  void register(Observer o);
  void deregister(Observer o);
  void notifyObservers();
}
class StockMarket implements Stock {
  private List<Observer> observers = new ArrayList<>();
  private float stockPrice;
  public void setStockPrice(float price) {
      this.stockPrice = price;
      notifyObservers();
  }
  public void register(Observer o) {
```

```
        observers.add(o);
    }
    public void deregister(Observer o) {
        observers.remove(o);
    }
    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(stockPrice);
        }
    }
}
class MobileApp implements Observer {
    public void update(float price) {
        System.out.println("MobileApp - Stock price updated: " + price);
} }
class WebApp implements Observer {
    public void update(float price) {
        System.out.println("WebApp - Stock price updated: " + price);
} }
public class ObserverPatternExample {
    public static void main(String[] args) {
        StockMarket stockMarket = new StockMarket();
        Observer mobileApp = new MobileApp();
        Observer webApp = new WebApp();
        stockMarket.register(mobileApp);
        stockMarket.register(webApp);
        stockMarket.setStockPrice(100.5f);
        stockMarket.setStockPrice(102.75f);
} }
```

Output:

```
Problems  Javadoc  Declaration  Console ×
<terminated> ObserverPatternExample [Java Application] C:\Users\ze
MobileApp - Stock price updated: 100.5
WebApp - Stock price updated: 100.5
MobileApp - Stock price updated: 102.75
WebApp - Stock price updated: 102.75
```

Exercise 8: Implementing the Strategy Pattern

Code:

```
package week_1.Design_Patterns_Principles;
interface PaymentStrategy {
void pay(double amount);
}
class CreditCardPayment implements PaymentStrategy {
public void pay(double amount) {
```

```
    System.out.println("Paid " + amount + " using Credit Card.");
}
}
class PayPalPayment implements PaymentStrategy {
public void pay(double amount) {
    System.out.println("Paid " + amount + " using PayPal.");
} }
class PaymentContext {
private PaymentStrategy strategy;
public void setPaymentStrategy(PaymentStrategy strategy) {
    this.strategy = strategy;
}
public void payAmount(double amount) {
    strategy.pay(amount);
} }
public class StrategyPatternExample {
public static void main(String[] args) {
    PaymentContext context = new PaymentContext();
    context.setPaymentStrategy(new CreditCardPayment());
    context.payAmount(250.0);
    context.setPaymentStrategy(new PayPalPayment());
    context.payAmount(500.0);
} }
```

Output:

```
Problems  Javadoc  Declaration  Console ×
<terminated> StrategyPatternExample [Java Application] C:\Users\ze
Paid 250.0 using Credit Card.
Paid 500.0 using PayPal.
```

Example 9: Implementing the Command Pattern

Code:

```
package week_1.Design_Patterns_Principles;
interface Command {
void execute();
}class Light {
public void turnOn() {
    System.out.println("Light is ON.");
}
public void turnOff() {
    System.out.println("Light is OFF.");
} }
class LightOnCommand implements Command {
private Light light;
public LightOnCommand(Light light) {
    this.light = light;
}
public void execute() {
```

```
    light.turnOn();
} }
class LightOffCommand implements Command {
private Light light;
public LightOffCommand(Light light) {
    this.light = light;
}
public void execute() {
    light.turnOff();
} }
class RemoteControl {
private Command command;
public void setCommand(Command command) {
    this.command = command;
}
public void pressButton() {
    command.execute();
} }
public class CommandPatternExample {
public static void main(String[] args) {
    Light light = new Light();
    Command onCommand = new LightOnCommand(light);
    Command offCommand = new LightOffCommand(light);
    RemoteControl remote = new RemoteControl();
    remote.setCommand(onCommand);
    remote.pressButton();
    remote.setCommand(offCommand);
    remote.pressButton();
} }
```

Output:

```
Problems  Javadoc  Declaration  Console ×
<terminated> CommandPatternExample [Java Application] C:
Light is ON.
Light is OFF.
```

Exercise 10: Implementing the MVC pattern

Code:

```
class Student {
private String name;
private String id;
private String grade;
public Student(String name, String id, String grade) {
    this.name = name;
    this.id = id;
    this.grade = grade;
}
public String getName() { return name; }
```

```java
public String getId() { return id; }
public String getGrade() { return grade; }
public void setName(String name) { this.name = name; }
public void setGrade(String grade) { this.grade = grade; }
}
class StudentView {
public void displayStudentDetails(Student student) {
   System.out.println("Student: ");
   System.out.println("Name: " + student.getName());
   System.out.println("ID: " + student.getId());
   System.out.println("Grade: " + student.getGrade());
} }
class StudentController {
private Student model;
private StudentView view;
public StudentController(Student model, StudentView view) {
   this.model = model;
   this.view = view;
}
public void setStudentName(String name) {
   model.setName(name);
}
public void setStudentGrade(String grade) {
   model.setGrade(grade);
}
public void updateView() {
   view.displayStudentDetails(model);
} }
public class MVCPatternExample {
public static void main(String[] args) {
   Student model = new Student("Varsha", "226", "A");
   StudentView view = new StudentView();
   StudentController controller = new StudentController(model, view);
   controller.updateView();
   controller.setStudentName("Varsha R");
   controller.setStudentGrade("B+");
   controller.updateView();
} }
```

Output:

```
Student:
Name: Varsha
ID: 226
Grade: A
Student:
Name: Varsha R
ID: 226
Grade: B+
```

Exercise 11: Implementing Dependency Injection

Code:

```
package week_1.Design_Patterns_Principles;
class Customer {
private int id;
private String name;
private String email;
public Customer(int id, String name, String email) {
   this.id = id;
   this.name = name;
   this.email = email;
}
public int getId() { return id; }
public String getName() { return name; }
public String getEmail() { return email; }
}
interface CustomerRepository {
Customer findCustomerById(int id);
}
class CustomerRepositoryImpl implements CustomerRepository {
public Customer findCustomerById(int id) {
   if (id == 101) {
      return new Customer(101, "Varsha", "varsha123@gmail.com");
   } else if (id == 102) {
      return new Customer(102, "Ravi", "ravi123@gmail.com");
   } else {
      return null;
   }
}
}
class CustomerService {
private CustomerRepository repository;
public CustomerService(CustomerRepository repository) {
   this.repository = repository;
}
public void getCustomerDetails(int id) {
   Customer customer = repository.findCustomerById(id);
   if (customer != null) {
      System.out.println("Customer ID: " + customer.getId());
      System.out.println("Customer Name: " + customer.getName());
      System.out.println("Customer Email: " + customer.getEmail());
   } else {
      System.out.println("Customer not found with ID: " + id);
   }
}
}
public class DependencyInjectionExample {
public static void main(String[] args) {
   CustomerRepository repository = new CustomerRepositoryImpl();
   CustomerService service = new CustomerService(repository);
   service.getCustomerDetails(101);
```

```
    System.out.println();
    service.getCustomerDetails(102);
    System.out.println();
    service.getCustomerDetails(999);
}
}
```

Output:

```
Customer ID: 101
Customer Name: Varsha
Customer Email: varsha123@gmail.com

Customer ID: 102
Customer Name: Ravi
Customer Email: ravi123@gmail.com

Customer not found with ID: 999
```