**Exercise 1: Problem Statement on Design patterns**

**Come up creatively with six different use cases to demonstrate your understanding of the following software design patterns by coding the same.**

**Two use cases to demonstrate two behavioural design pattern.**

**Two use cases to demonstrate two creational design pattern.**

**Two use cases to demonstrate two structural design pattern.**

**2. 3. 1.**

**Code:**

**1. Observer Pattern**

**Use Case: Notification System**

**java**

**Copy code**

**import java.util.ArrayList;**

**import java.util.List;**

**// Subject**

**interface Subject {**

**void addObserver(Observer observer);**

**void removeObserver(Observer observer);**

**void notifyObservers(String message);**

**}**

**class ConcreteSubject implements Subject {**

**private List<Observer> observers = new ArrayList<>();**

**@Override**

**public void addObserver(Observer observer) {**

**observers.add(observer);**

**}**

**@Override**

**public void removeObserver(Observer observer) {**

**observers.remove(observer);**

**}**

**@Override**

**public void notifyObservers(String message) {**

**for (Observer observer : observers) {**

**observer.update(message);**

**}**

**}**

**}**

**// Observer**

**interface Observer {**

**void update(String message);**

**}**

**class ConcreteObserver implements Observer {**

**private String name;**

**public ConcreteObserver(String name) {**

**this.name = name;**

**}**

**@Override**

**public void update(String message) {**

**System.out.println(name + " received message: " + message);**

**}**

**}**

**// Usage**

**public class ObserverPatternDemo {**

**public static void main(String[] args) {**

**ConcreteSubject subject = new ConcreteSubject();**

**Observer observer1 = new ConcreteObserver("Observer1");**

**Observer observer2 = new ConcreteObserver("Observer2");**

**subject.addObserver(observer1);**

**subject.addObserver(observer2);**

**subject.notifyObservers("Hello Observers!");**

**}**

**}**

**2. Strategy Pattern**

**Use Case: Payment Processing**

**java**

**Copy code**

**// Strategy**

**interface PaymentStrategy {**

**void pay(int amount);**

**}**

**class CreditCardPayment implements PaymentStrategy {**

**@Override**

**public void pay(int amount) {**

**System.out.println("Paid " + amount + " using Credit Card");**

**}**

**}**

**class PayPalPayment implements PaymentStrategy {**

**@Override**

**public void pay(int amount) {**

**System.out.println("Paid " + amount + " using PayPal");**

**}**

**}**

**// Context**

**class PaymentContext {**

**private PaymentStrategy strategy;**

**public PaymentContext(PaymentStrategy strategy) {**

**this.strategy = strategy;**

**}**

**public void executePayment(int amount) {**

**strategy.pay(amount);**

**}**

**}**

**// Usage**

**public class StrategyPatternDemo {**

**public static void main(String[] args) {**

**PaymentContext context = new PaymentContext(new CreditCardPayment());**

**context.executePayment(100);**

**context = new PaymentContext(new PayPalPayment());**

**context.executePayment(200);**

**}**

**}**

**Creational Design Patterns**

**1. Singleton Pattern**

**Use Case: Configuration Manager**

**java**

**Copy code**

**// Singleton**

**public class ConfigurationManager {**

**private static ConfigurationManager instance;**

**private Map<String, String> config = new HashMap<>();**

**private ConfigurationManager() { }**

**public static synchronized ConfigurationManager getInstance() {**

**if (instance == null) {**

**instance = new ConfigurationManager();**

**}**

**return instance;**

**}**

**public void set(String key, String value) {**

**config.put(key, value);**

**}**

**public String get(String key) {**

**return config.get(key);**

**}**

**}**

**// Usage**

**public class SingletonPatternDemo {**

**public static void main(String[] args) {**

**ConfigurationManager config1 = ConfigurationManager.getInstance();**

**config1.set("db\_host", "localhost");**

**ConfigurationManager config2 = ConfigurationManager.getInstance();**

**System.out.println(config2.get("db\_host")); // Outputs: localhost**

**}**

**}**

**2. Factory Method Pattern**

**Use Case: Document Creation**

**java**

**Copy code**

**// Product**

**interface Document {**

**void read();**

**}**

**class PDFDocument implements Document {**

**@Override**

**public void read() {**

**System.out.println("Reading PDF document");**

**}**

**}**

**class WordDocument implements Document {**

**@Override**

**public void read() {**

**System.out.println("Reading Word document");**

**}**

**}**

**// Creator**

**abstract class DocumentFactory {**

**public abstract Document createDocument();**

**}**

**class PDFDocumentFactory extends DocumentFactory {**

**@Override**

**public Document createDocument() {**

**return new PDFDocument();**

**}**

**}**

**class WordDocumentFactory extends DocumentFactory {**

**@Override**

**public Document createDocument() {**

**return new WordDocument();**

**}**

**}**

**// Usage**

**public class FactoryMethodPatternDemo {**

**public static void main(String[] args) {**

**DocumentFactory factory = new PDFDocumentFactory();**

**Document document = factory.createDocument();**

**document.read();**

**factory = new WordDocumentFactory();**

**document = factory.createDocument();**

**document.read();**

**}**

**}**

**Structural Design Patterns**

**1. Adapter Pattern**

**Use Case: Legacy System Integration**

**java**

**Copy code**

**// Target**

**interface NewSystem {**

**String newMethod();**

**}**

**// Adaptee**

**class OldSystem {**

**public String oldMethod() {**

**return "Data from old system";**

**}**

**}**

**// Adapter**

**class Adapter implements NewSystem {**

**private OldSystem oldSystem;**

**public Adapter(OldSystem oldSystem) {**

**this.oldSystem = oldSystem;**

**}**

**@Override**

**public String newMethod() {**

**return oldSystem.oldMethod();**

**}**

**}**

**// Usage**

**public class AdapterPatternDemo {**

**public static void main(String[] args) {**

**OldSystem oldSystem = new OldSystem();**

**NewSystem adapter = new Adapter(oldSystem);**

**System.out.println(adapter.newMethod()); // Outputs: Data from old system**

**}**

**}**

**2. Decorator Pattern**

**Use Case: Adding Features to a Car**

**java**

**Copy code**

**// Component**

**interface Car {**

**String getDescription();**

**double cost();**

**}**

**class BasicCar implements Car {**

**@Override**

**public String getDescription() {**

**return "Basic Car";**

**}**

**@Override**

**public double cost() {**

**return 10000;**

**}**

**}**

**// Decorator**

**abstract class CarDecorator implements Car {**

**protected Car car;**

**public CarDecorator(Car car) {**

**this.car = car;**

**}**

**}**

**class LeatherSeats extends CarDecorator {**

**public LeatherSeats(Car car) {**

**super(car);**

**}**

**@Override**

**public String getDescription() {**

**return car.getDescription() + ", Leather Seats";**

**}**

**@Override**

**public double cost() {**

**return car.cost() + 2000;**

**}**

**}**

**class Sunroof extends CarDecorator {**

**public Sunroof(Car car) {**

**super(car);**

**}**

**@Override**

**public String getDescription() {**

**return car.getDescription() + ", Sunroof";**

**}**

**@Override**

**public double cost() {**

**return car.cost() + 1500;**

**}**

**}**

**// Usage**

**public class DecoratorPatternDemo {**

**public static void main(String[] args) {**

**Car car = new BasicCar();**

**System.out.println(car.getDescription()); // Outputs: Basic Car**

**System.out.println(car.cost()); // Outputs: 10000**

**car = new LeatherSeats(car);**

**car = new Sunroof(car);**

**System.out.println(car.getDescription()); // Outputs: Basic Car, Leather Seats, Sunroof**

**System.out.println(car.cost()); // Outputs: 11500**

**}**

**}**

**Astronaut Daily Schedule Organizer Programming Exercise**

**Problem Statement**

**Design and implement a console-based application that helps astronauts organize their daily schedules. The application should allow users to add, remove, and view daily tasks. Each task will have a description, start time, end time, and priority level. The intent behind this problem statement is to evaluate your ability to implement a basic CRUD (Create, Read, Update, Delete) application, manage data efficiently, and apply best coding practices.**

**Functional Requirements**

**Mandatory Requirements**

**1.**

**Add a new task with description, start time, end time, and priority level.**

**2.**

**Remove an existing task.**

**3.**

**View all tasks sorted by start time.**

**4.**

**Validate that new tasks do not overlap with existing tasks.**

**5.**

**Provide appropriate error messages for invalid operations.**

**Optional Requirements**

**1.**

**Edit an existing task.**

**2.**

**Mark tasks as completed.**

**3.**

**View tasks for a specific priority level.**

**Non-functional Requirements**

**1.**

**The application should handle exceptions gracefully.**

**2.**

**Ensure the application is optimized for performance.**

**3.**

**Implement a logging mechanism for tracking application usage and errors.**

**Key Focus**

**Design Patterns to be used**

**1.**

**Singleton Pattern:**

**Ensure there is only one instance of the schedule manager.**

**2.**

**Factory Pattern:**

**Use a factory to create task objects.**

**3.**

**Observer Pattern:**

**Notify users of task conflicts or updates.**

**Detailed Instructions**

**1.**

**Use the Singleton Pattern to create a ScheduleManager class that manages all tasks.**

**2.**

**Implement a TaskFactory class to create Task objects.**

**3.**

**Use the Observer Pattern to alert users if a new task conflicts with an existing one.**

**Possible Inputs and Corresponding Outputs**

**Positive Cases**

**1.**

**Input:**

**Add Task("Morning Exercise", "07:00", "08:00", "High")**

**Output:**

**Task added successfully. No conflicts.**

**2.**

**Input:**

**Add Task("Team Meeting", "09:00", "10:00", "Medium")**

**Output:**

**Task added successfully. No conflicts.**

**3.**

**Input:**

**View Tasks**

**Output:**

**a.**

**07:00 - 08:00: Morning Exercise [High]**

**b.**

**09:00 - 10:00: Team Meeting [Medium]**

**4.**

**Input:**

**Remove Task("Morning Exercise")**

**Output:**

**Task removed successfully.**

**5.**

**Input:**

**Add Task("Lunch Break", "12:00", "13:00", "Low")**

**Output:**

**Task added successfully. No conflicts.**

**Negative Cases**

**1.**

**Input:**

**Add Task("Training Session", "09:30", "10:30", "High")**

**Output:**

**Error: Task conflicts with existing task "Team Meeting".**

**2.**

**Input:**

**Remove Task("Non-existent Task")**

**Output:**

**Error: Task not found.**

**3.**

**Input:**

**Add Task("Invalid Time Task", "25:00", "26:00", "Low")**

**Output:**

**Error: Invalid time format.**

**4.**

**Input:**

**Add Task("Overlap Task", "08:30", "09:30", "Medium")**

**Output:**

**Error: Task conflicts with existing task "Team Meeting".**

**5.**

**Input:**

**View Tasks (when no tasks exist)**

**Output:**

**No tasks scheduled for the day.**

**Evaluation**

**1.**

**Code Quality:**

**Adherence to best practices, use of design patterns, SOLID principles, and OOP.**

**2.**

**Functionality:**

**All mandatory requirements implemented correctly.**

**3.**

**Error Handling:**

**Graceful handling of all errors and edge cases.**

**4.**

**Performance:**

**Code is optimized for performance.**

**5.**

**Explanation:**

**Candidate's ability to walk through the code and explain design decisions and logic.**

**6.**

**Documentation:**

**Code is well-documented, and usage instructions are clear.**

**The goal of this exercise is to assess the candidate's coding skills, understanding of design patterns, and ability to produce high-quality, maintainable code.**

**Solution:**

**1. Task Class**

java

Copy code

import java.time.LocalTime;

public class Task {

private String description;

private LocalTime startTime;

private LocalTime endTime;

private String priorityLevel;

public Task(String description, LocalTime startTime, LocalTime endTime, String priorityLevel) {

this.description = description;

this.startTime = startTime;

this.endTime = endTime;

this.priorityLevel = priorityLevel;

}

public String getDescription() {

return description;

}

public LocalTime getStartTime() {

return startTime;

}

public LocalTime getEndTime() {

return endTime;

}

public String getPriorityLevel() {

return priorityLevel;

}

@Override

public String toString() {

return startTime + " - " + endTime + ": " + description + " [" + priorityLevel + "]";

}

}

**2. TaskFactory Class**

java

Copy code

import java.time.LocalTime;

public class TaskFactory {

public static Task createTask(String description, LocalTime startTime, LocalTime endTime, String priorityLevel) {

return new Task(description, startTime, endTime, priorityLevel);

}

}

**3. Observer Interface**

java

Copy code

public interface Observer {

void update(String message);

}

**4. TaskConflictObserver Class**

java

Copy code

public class TaskConflictObserver implements Observer {

@Override

public void update(String message) {

System.out.println("Conflict Notification: " + message);

}

}

**5. ScheduleManager Class (Singleton)**

java

Copy code

import java.time.LocalTime;

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;

public class ScheduleManager {

private static ScheduleManager instance;

private List<Task> tasks;

private List<Observer> observers;

private ScheduleManager() {

tasks = new ArrayList<>();

observers = new ArrayList<>();

}

public static synchronized ScheduleManager getInstance() {

if (instance == null) {

instance = new ScheduleManager();

}

return instance;

}

public void addObserver(Observer observer) {

observers.add(observer);

}

public void removeObserver(Observer observer) {

observers.remove(observer);

}

public void addTask(String description, LocalTime startTime, LocalTime endTime, String priorityLevel) {

Task newTask = TaskFactory.createTask(description, startTime, endTime, priorityLevel);

if (isConflict(newTask)) {

notifyObservers("Task conflicts with an existing task: " + description);

return;

}

tasks.add(newTask);

tasks.sort((t1, t2) -> t1.getStartTime().compareTo(t2.getStartTime()));

System.out.println("Task added successfully. No conflicts.");

}

public void removeTask(String description) {

Task taskToRemove = tasks.stream()

.filter(task -> task.getDescription().equals(description))

.findFirst()

.orElse(null);

if (taskToRemove != null) {

tasks.remove(taskToRemove);

System.out.println("Task removed successfully.");

} else {

System.out.println("Error: Task not found.");

}

}

public void viewTasks() {

if (tasks.isEmpty()) {

System.out.println("No tasks scheduled for the day.");

} else {

tasks.forEach(System.out::println);

}

}

private boolean isConflict(Task newTask) {

for (Task existingTask : tasks) {

if (newTask.getStartTime().isBefore(existingTask.getEndTime()) &&

newTask.getEndTime().isAfter(existingTask.getStartTime())) {

return true;

}

}

return false;

}

private void notifyObservers(String message) {

for (Observer observer : observers) {

observer.update(message);

}

}

}

**6. Main Class**

java

Copy code

import java.time.LocalTime;

public class Main {

public static void main(String[] args) {

ScheduleManager manager = ScheduleManager.getInstance();

TaskConflictObserver conflictObserver = new TaskConflictObserver();

manager.addObserver(conflictObserver);

manager.addTask("Morning Exercise", LocalTime.of(7, 0), LocalTime.of(8, 0), "High");

manager.addTask("Team Meeting", LocalTime.of(9, 0), LocalTime.of(10, 0), "Medium");

manager.viewTasks();

manager.addTask("Training Session", LocalTime.of(9, 30), LocalTime.of(10, 30), "High"); // Conflict

manager.removeTask("Non-existent Task"); // Error

manager.removeTask("Morning Exercise");

manager.addTask("Lunch Break", LocalTime.of(12, 0), LocalTime.of(13, 0), "Low");

manager.viewTasks();

}

}