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.

1. Two use cases to demonstrate two behavioural design pattern.
2. Two use cases to demonstrate two creational design pattern.
3. Two use cases to demonstrate two structural design pattern.

1.**Behavioral Design Patterns**:

a) **Observer Participant:**

*"**Weather Station Notification System:** A weather station that notifies different display units (like a phone app, a billboard, and a website) about weather changes."*

- In this scenario, the **WeatherStation** class notifies registered observers, like **PhoneDisplay** and **WebsiteDisplay**, whenever there's a change in weather data. This design allows for easy addition or removal of displays and ensures that all devices are updated simultaneously, making it highly flexible and responsive to real-time changes.

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

interface Observer {
    void update(float temperature, float humidity);
}

class WeatherStation {
    private List<Observer> observers;
    private float temperature;
    private float humidity;

    public WeatherStation() {
        observers = new ArrayList<>();
    }

    public void addObserver(Observer observer) {
        observers.add(observer);
    }
```

```
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature, humidity);
        }
    }

    public void setWeather(float temperature, float humidity) {
        this.temperature = temperature;
        this.humidity = humidity;
        notifyObservers();
    }
}

class PhoneDisplay implements Observer {
    private String name;

    public PhoneDisplay(String name) {
        this.name = name;
    }

    @Override
    public void update(float temperature, float humidity) {
        System.out.println(name + " - Phone Display: Temperature = " +
temperature + ", Humidity = " + humidity);
    }
}

class WebsiteDisplay implements Observer {
    private String name;

    public WebsiteDisplay(String name) {
        this.name = name;
    }

    @Override
    public void update(float temperature, float humidity) {
        System.out.println(name + " - Website Display: Temperature = " +
temperature + ", Humidity = " + humidity);
    }
}

public class ObserverPatternWeatherStation {
    public static void main(String[] args) {
```

```
        WeatherStation weatherStation = new WeatherStation();

        Observer phoneDisplay = new PhoneDisplay("Phone App");
        Observer websiteDisplay = new WebsiteDisplay("Weather Website");

        weatherStation.addObserver(phoneDisplay);
        weatherStation.addObserver(websiteDisplay);

        weatherStation.setWeather(30.0f, 70.0f);  // Simulating a weather
change
    }
}
```

b) <u>Chain of Responsibility Pattern:</u>

Customer Support System: *A customer support system where the request is passed along a chain of handlers (like support agents, team leaders, and managers) based on the severity of the issue.*

- In this system, each handler class, like **AgentSupport, TeamLeadSupport,** and **ManagerSupport**, checks whether it can handle a specific request. If it cannot, the request is passed along the chain until an appropriate handler is found. This structure provides the flexibility to add new levels of support easily and organizes the handling of issues efficiently, ensuring that customers receive the help they need promptly.

```
abstract class SupportHandler {
    protected SupportHandler nextHandler;

    public void setNextHandler(SupportHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    public abstract void handleRequest(String message, int level);
}

class AgentSupport extends SupportHandler {
    @Override
    public void handleRequest(String message, int level) {
        if (level == 1) {
            System.out.println("Agent handling request: " + message);
        } else if (nextHandler != null) {
            nextHandler.handleRequest(message, level);
```

```
        }
    }
}

class TeamLeadSupport extends SupportHandler {
    @Override
    public void handleRequest(String message, int level) {
        if (level == 2) {
            System.out.println("Team Lead handling request: " + message);
        } else if (nextHandler != null) {
            nextHandler.handleRequest(message, level);
        }
    }
}

class ManagerSupport extends SupportHandler {
    @Override
    public void handleRequest(String message, int level) {
        if (level >= 3) {
            System.out.println("Manager handling request: " + message);
        } else if (nextHandler != null) {
            nextHandler.handleRequest(message, level);
        }
    }
}

public class ChainOfResponsibilityCustomerSupport {
    public static void main(String[] args) {
        SupportHandler agent = new AgentSupport();
        SupportHandler teamLead = new TeamLeadSupport();
        SupportHandler manager = new ManagerSupport();

        agent.setNextHandler(teamLead);
        teamLead.setNextHandler(manager);

        agent.handleRequest("Simple issue", 1);  // Handled by agent
        agent.handleRequest("Intermediate issue", 2);  // Handled by team lead
        agent.handleRequest("Critical issue", 3);  // Handled by manager
    }
}
```

2.Creational Design Patterns

a) Singleton Pattern:

Printer Spooler *A printer spooler that ensures only one instance of the spooler manages all print jobs in the system.*

- The **PrinterSpooler** class controls its own instantiation, allowing multiple print jobs to be queued through a single access point. This design prevents conflicts between print jobs and ensures efficient resource management by centralizing control, making it ideal for environments where print requests can occur simultaneously.

```
class PrinterSpooler {
    private static PrinterSpooler instance;

    private PrinterSpooler() {
        System.out.println("Printer Spooler initialized");
    }

    public static PrinterSpooler getInstance() {
        if (instance == null) {
            instance = new PrinterSpooler();
        }
        return instance;
    }

    public void printJob(String document) {
        System.out.println("Printing: " + document);
    }
}

public class SingletonPrinterSpoolerExample {
    public static void main(String[] args) {
        PrinterSpooler spooler = PrinterSpooler.getInstance();
        spooler.printJob("Document1.pdf");
    }
}
```

b) <u>Abstract Factory Pattern:</u>

GUI Framework Factory *A system that creates either Windows or Mac GUI components (buttons, text fields) depending on the platform.*

- Here, the **GUIFactory** interface defines methods for creating various UI components, with concrete factories implementing these methods for each specific platform. This approach facilitates easy swapping of component implementations depending on the target platform, allowing developers to maintain a consistent user

experience across different environments without modifying the core application code.

```
interface Button {
    void paint();
}

interface TextField {
    void render();
}

interface GUIFactory {
    Button createButton();
    TextField createTextField();
}

class WindowsButton implements Button {
    @Override
    public void paint() {
        System.out.println("Painting Windows Button");
    }
}

class WindowsTextField implements TextField {
    @Override
    public void render() {
        System.out.println("Rendering Windows Text Field");
    }
}

class MacButton implements Button {
    @Override
    public void paint() {
        System.out.println("Painting Mac Button");
    }
}

class MacTextField implements TextField {
    @Override
    public void render() {
        System.out.println("Rendering Mac Text Field");
    }
}

class WindowsFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new WindowsButton();
```

```
    }

    @Override
    public TextField createTextField() {
        return new WindowsTextField();
    }
}

class MacFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new MacButton();
    }

    @Override
    public TextField createTextField() {
        return new MacTextField();
    }
}

public class AbstractFactoryGUIExample {
    public static void main(String[] args) {
        GUIFactory factory;

        String os = "Windows";  // This can be dynamic based on the user's
platform

        if (os.equals("Windows")) {
            factory = new WindowsFactory();
        } else {
            factory = new MacFactory();
        }

        Button button = factory.createButton();
        TextField textField = factory.createTextField();

        button.paint();
        textField.render();
    }
}
```

3.Structural Design Patterns

a) <u>Composite Pattern:</u>

Company Hierarchy *A company hierarchy where employees are either managers (having subordinates) or simple workers (with no subordinates).*

- The **Employee** interface is implemented by **Worker** classes (representing individual employees) and Manager classes (which can manage multiple workers). This structure simplifies code by allowing clients to treat individual employees and groups of employees interchangeably, thus enhancing the organization and management of complex hierarchies.

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

interface Employee {
    void showDetails();
}

class Worker implements Employee {
    private String name;
    private String position;

    public Worker(String name, String position) {
        this.name = name;
        this.position = position;
    }

    @Override
    public void showDetails() {
        System.out.println(name + " works as " + position);
    }
}

class Manager implements Employee {
    private String name;
    private List<Employee> subordinates = new ArrayList<>();

    public Manager(String name) {
        this.name = name;
    }

    public void addEmployee(Employee employee) {
        subordinates.add(employee);
    }

    @Override
    public void showDetails() {
        System.out.println(name + " manages:");
        for (Employee employee : subordinates) {
            employee.showDetails();
        }
```

```
    }
}

public class CompositePatternCompanyHierarchy {
    public static void main(String[] args) {
        Employee worker1 = new Worker("John", "Developer");
        Employee worker2 = new Worker("Emily", "Designer");

        Manager manager = new Manager("Alice");
        manager.addEmployee(worker1);
        manager.addEmployee(worker2);

        manager.showDetails();
    }
}
```

b) Bridge Pattern:

Remote Control for Devices *: A remote control that works for different devices (like TVs, Radios) but allows each device to have different implementations (like turning on/off, volume control).*

- In this scenario, the **RemoteControl** class provides a high-level interface for controlling devices, while concrete device classes implement the specific actions, like turning on/off and adjusting volume. This design allows for flexibility in extending both remote and device functionalities independently, reducing complexity in client code and facilitating the addition of new device types or remote controls without disrupting existing implementations.

```
interface Device {
    void turnOn();
    void turnOff();
    void setVolume(int volume);
}

class TV implements Device {
    @Override
    public void turnOn() {
        System.out.println("TV is ON");
    }

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

```
    }

    @Override
    public void setVolume(int volume) {
        System.out.println("TV volume set to " + volume);
    }
}

class Radio implements Device {
    @Override
    public void turnOn() {
        System.out.println("Radio is ON");
    }

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

    @Override
    public void setVolume(int volume) {
        System.out.println("Radio volume set to " + volume);
    }
}

abstract class RemoteControl {
    protected Device device;

    public RemoteControl(Device device) {
        this.device = device;
    }

    public abstract void turnOn();

    public abstract void turnOff();

    public abstract void setVolume(int volume);
}

class AdvancedRemote extends RemoteControl {
    public AdvancedRemote(Device device) {
        super(device);
    }

    @Override
    public void turnOn() {
        device.turnOn();
    }
```

```
    @Override
    public void turnOff() {
        device.turnOff();
    }

    @Override
    public void setVolume(int volume) {
        device.setVolume(volume);
    }
}

public class BridgePatternRemoteControlExample {
    public static void main(String[] args) {
        Device tv = new TV();
        RemoteControl tvRemote = new AdvancedRemote(tv);

        tvRemote.turnOn();
        tvRemote.setVolume(10);
        tvRemote.turnOff();

        Device radio = new Radio();
        RemoteControl radioRemote = new AdvancedRemote(radio);

        radioRemote.turnOn();
        radioRemote.setVolume(5);
        radioRemote.turnOff();
    }
}
```