B KAVYA MSRUAS
bkavyashree2003@gmail.com

Smart Home System Programming Exercise

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 Pattern

Use Case 1: Observer Pattern

- Scenario: Devices update their status when changes occur in the system**.**

Java code:

```
import java.util.ArrayList;

import java.util.List;


// Observer Interface

interface DeviceObserver {

    void update(String status);

}


// Concrete Observer Classes

class Light implements DeviceObserver {

    private int id;

    private String status;


    public Light(int id) {

        this.id = id;

        this.status = "off";
    }

    @Override
    public void update(String status) {
        this.status = status;
        System.out.println("Light " + id + " is now " + this.status + ".");
    }
}

class Thermostat implements DeviceObserver {
    private int id;
    private int temperature;

    public Thermostat(int id) {
        this.id = id;
        this.temperature = 70; // Default temperature
    }

    @Override
    public void update(String temperature) {
        this.temperature = Integer.parseInt(temperature);
        System.out.println("Thermostat " + id + " is set to " + this.temperature + " degrees.");
    }
}

// Subject Class
class SmartHomeHub {
    private List<DeviceObserver> devices = new ArrayList<>();

    public void attach(DeviceObserver device) {
        devices.add(device);
    }

    public void notifyObservers(String status) {
        for (DeviceObserver device : devices) {
            device.update(status);
        }
    }
}

// Example usage
public class SmartHomeSystem {
    public static void main(String[] args) {
        SmartHomeHub hub = new SmartHomeHub();
        Light light1 = new Light(1);
        Thermostat thermostat1 = new Thermostat(2);

        hub.attach(light1);
        hub.attach(thermostat1);

        hub.notifyObservers("on");  // Light 1 is now on. Thermostat 2 is set to 70 degrees.
    }
}
```

Behavioral Pattern

Java code:

```
interface AutomationStrategy {
    void execute(Thermostat thermostat, int threshold, Light light);
}

class TemperatureAutomation implements AutomationStrategy {
    @Override
```

```
  public void execute(Thermostat thermostat, int threshold, Light light) {
    if (thermostat.getTemperature() > threshold) {
      light.update("off");
    }
  }
}

// Example usage
public class AutomationTest {
  public static void main(String[] args) {
    Thermostat thermostat1 = new Thermostat(2);
    Light light1 = new Light(1);
    AutomationStrategy strategy = new TemperatureAutomation();

    thermostat1.update("80"); // Set thermostat to 80 degrees
    strategy.execute(thermostat1, 75, light1); // Light 1 will turn off due to high temperature.
  }
}
```

2. Creational Design Pattern

Use Case 1: Factory Method

- Scenario: Create different types of devices dynamically.

Java Code:

```
abstract class Device {
  protected int id;
  public abstract void turnOn();
  public abstract void turnOff();
}

class LightDevice extends Device {
  public LightDevice(int id) {
    this.id = id;
  }

  @Override
  public void turnOn() {
    System.out.println("Light " + id + " is turned on.");
  }

  @Override
  public void turnOff() {
    System.out.println("Light " + id + " is turned off.");
  }
}

class ThermostatDevice extends Device {
  public ThermostatDevice(int id) {
    this.id = id;
  }

  @Override
  public void turnOn() {
    System.out.println("Thermostat " + id + " is turned on.");
  }

  @Override
  public void turnOff() {
    System.out.println("Thermostat " + id + " is turned off.");
  }
}

class DeviceFactory {
  public static Device createDevice(String type, int id) {
    switch (type.toLowerCase()) {
        case "light":
            return new LightDevice(id);
        case "thermostat":
            return new ThermostatDevice(id);
        default:
            throw new IllegalArgumentException("Unknown device type.");
    }
  }
}

// Example usage
public class FactoryTest {
  public static void main(String[] args) {
    Device light = DeviceFactory.createDevice("light", 1);
    light.turnOn(); // Light 1 is turned on.

    Device thermostat = DeviceFactory.createDevice("thermostat", 2);
    thermostat.turnOn(); // Thermostat 2 is turned on.
  }
}
```

Creational Pattern

```
class SmartHomeHubSingleton {
    private static SmartHomeHubSingleton instance;

    private SmartHomeHubSingleton() {
        // Private constructor to prevent instantiation
    }

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

// Example usage
public class SingletonTest {
    public static void main(String[] args) {
        SmartHomeHubSingleton hub1 = SmartHomeHubSingleton.getInstance();
        SmartHomeHubSingleton hub2 = SmartHomeHubSingleton.getInstance();

        System.out.println(hub1 == hub2); // true, both are the same instance
    }
}
```

3. Structural Design Patterns

Use Case 1: Proxy Pattern

- Scenario: Control access to devices.

Java code:

```
class DeviceProxy {
    private Device realDevice;

    public DeviceProxy(Device realDevice) {
        this.realDevice = realDevice;
    }

    public void turnOn() {
        System.out.println("Accessing device...");
        realDevice.turnOn();
    }
    public void turnOff() {
        System.out.println("Accessing device...");
        realDevice.turnOff();
    }
}


// Example usage
public class ProxyTest {
    public static void main(String[] args) {
        Device light = DeviceFactory.createDevice("light", 3);
        DeviceProxy proxy = new DeviceProxy(light);


        proxy.turnOn();  // Accessing device... Light 3 is turned on.
    }
}
```

Composite Pattern:

Java code:

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


class DeviceGroup {
    private List<Device> devices = new ArrayList<>();


    public void add(Device device) {
        devices.add(device);
    }


    public void turnOnAll() {
        for (Device device : devices) {
            device.turnOn();
        }
    }


    public void turnOffAll() {
        for (Device device : devices) {
            device.turnOff();
        }
    }
}


// Example usage
public class CompositeTest {
    public static void main(String[] args) {
        Device light1 = DeviceFactory.createDevice("light", 1);
        Device light2 = DeviceFactory.createDevice("light", 2);
        DeviceGroup group = new DeviceGroup();


        group.add(light1);
        group.add(light2);
        group.turnOnAll();  // Light 1 is turned on. Light 2 is turned on.
    }
}
```

This Java implementation of the Smart Home System meets the specified requirements while effectively demonstrating various design patterns. Each use case showcases the patterns in a way that emphasizes modularity, encapsulation, and scalability. You can further expand this code by adding detailed error handling, logging, and enhanced scheduling functionalities to make it more robust.