

Question ID: Mediator_GhostWriter02

Attempt no. 2/3

Question:

1. Write mediator pattern in Java for smart home appliances.
2. Can you help me find code smells in the generated code?
3. Can you help me refactor the code?
4. Does the refactored code still use the mediator pattern?
5. Does the refactored code still have code smells?
6. Can you please write JUnit test for the code?
7. What is the test coverage of the JUnit test?

Replit Ghostwriter answers

Q1: Write mediator pattern in Java for smart home appliances.

```
interface Appliance {  
    public void turnOn();  
  
    public void turnOff();  
  
    public String getState();  
}  
  
class Fan implements Appliance {  
    private String state;  
  
    public void turnOn() {  
        this.state = "ON";  
        System.out.println("Fan turned ON");  
    }  
  
    public void turnOff() {  
        this.state = "OFF";  
        System.out.println("Fan turned OFF");  
    }  
  
    public String getState() {  
        return this.state;  
    }  
}
```

```
}
```

```
class Light implements Appliance {
```

```
    private String state;
```

```
    public void turnOn() {
```

```
        this.state = "ON";
```

```
        System.out.println("Light turned ON");
```

```
    }
```

```
    public void turnOff() {
```

```
        this.state = "OFF";
```

```
        System.out.println("Light turned OFF");
```

```
    }
```

```
    public String getState() {
```

```
        return this.state;
```

```
    }
```

```
}
```

```
interface Control {
```

```
    public void turnOn();
```

```
    public void turnOff();
```

```
    public String getState();
```

```
    public void setAppliance(Appliance appliance);
```

```
}
```

```
class SmartHomeControl implements Control {
```

```
    private Appliance appliance;
```

```
    public void turnOn() {
```

```
        appliance.turnOn();
```

```
    }
```

```
    public void turnOff() {
```

```
        appliance.turnOff();
```

```
    }
```

```
    public String getState() {
```

```
        return appliance.getState();
```

```
    }
```

```
    public void setAppliance(Appliance appliance) {
```

```

        this.appliance = appliance;
    }
}

class Mediator {
    private List<Control> controls = new ArrayList<Control>();

    public void followCommand(String command, Control control) {
        if (command.equalsIgnoreCase("TURNON")) {
            control.turnOn();
        } else if (command.equalsIgnoreCase("TURNOFF")) {
            control.turnOff();
        }
    }
    public void addControl(Control control) {
        controls.add(control);
    }
}

```

Q2: Can you help me find code smells in the generated code?

The generated code appears to be well-written and does not contain any major code smells. However, there are a few minor things that could be improved:

1. The public keyword is redundant in interface Appliance.
2. The Mediator class should have a private constructor to prevent accidental instantiation.
3. The Mediator class could benefit from additional methods for removing controls from the controls list and checking if a control is already in the list.
4. It might be more appropriate to use an enum for the state variable in Fan and Light instead of a string.

Overall, the generated code is well-structured and readable.

Q3: Can you help me refactor the code?

The generated code is already well-structured and readable, but here are a few suggestions for refactoring:

1. Rename interface Appliance to interface Device. The latter is more common terminology in the home automation context.
2. Replace private String state in both Fan and Light implementations with an enum.
3. Use the @Override annotation when implementing interface methods to improve code clarity.
4. Consider using a switch-case statement instead of if-else for followCommand() method.
5. Use the diamond operator <> when initializing the ArrayList.
6. Add input parameter validation to prevent null error in the SmartHomeControl class.

Here's the refactored code:

```
enum State {  
    ON, OFF  
}
```

```
interface Device {  
    void turnOn();  
  
    void turnOff();  
  
    State getState();  
}
```

```
class Fan implements Device {  
    private State state;  
  
    @Override  
    public void turnOn() {  
        this.state = State.ON;  
        System.out.println("Fan turned ON");  
    }  
  
    @Override  
    public void turnOff() {  
        this.state = State.OFF;  
        System.out.println("Fan turned OFF");  
    }  
  
    @Override
```

```
    public State getState() {  
        return this.state;  
    }  
}
```

```
class Light implements Device {  
    private State state;
```

```
    @Override  
    public void turnOn() {  
        this.state = State.ON;  
        System.out.println("Light turned ON");  
    }
```

```
    @Override  
    public void turnOff() {  
        this.state = State.OFF;  
        System.out.println("Light turned OFF");  
    }
```

```
    @Override  
    public State getState() {  
        return this.state;  
    }  
}
```

```
interface Control {  
    void turnOn();  
  
    void turnOff();  
  
    State getState();  
  
    void setDevice(Device device);  
}
```

```
class SmartHomeControl implements Control {  
    private Device device;
```

```

@Override
public void turnOn() {
    if (device != null) {
        device.turnOn();
    } else {
        System.out.println("Device not set");
    }
}

```

```

@Override
public void turnOff() {
    if (device != null) {
        device.turnOff();
    } else {
        System.out.println("Device not set");
    }
}

```

```

@Override
public State getState() {
    if (device != null) {
        return device.getState();
    } else {
        System.out.println("Device not set");
        return null;
    }
}

```

```

@Override
public void setDevice(Device device) {
    this.device = device;
}
}

```

```

class Mediator {
    private List<Control> controls = new ArrayList<>();

    public void followCommand(String command, Control control) {
        switch (command.toUpperCase()) {

```

```

        case "TURNON":
            control.turnOn();
            break;
        case "TURNOFF":
            control.turnOff();
            break;
        default:
            System.out.println("Invalid command");
            break;
    }
}

public void addControl(Control control) {
    controls.add(control);
}
}

```

I hope that helps!

Q4: Does the refactored code still use the mediator pattern? [also given the code]

The refactored code still uses the mediator pattern. The pattern is implemented in the Mediator class, which mediates commands between Control objects and devices (Device objects), which implement the Device interface with methods to turn on and off the device and retrieve its state. The SmartHomeController class acts as the concrete implementation of the Control interface.

Q5: Does the refactored code still have code smells?

Without knowing the requirements and context of the code, it is difficult to definitively say whether or not the refactored code still has code smells. However, at a high level, the code looks much cleaner and follows better encapsulation and SOLID principles than the original code.

Some potential improvements that could be made would be to add error handling for null reference of device in SmartHomeController methods (since it may cause NullPointerExceptions) and improve the naming of methods (e.g. followCommand could be refactored to something like mediateCommand). Overall, the code looks well-structured and easy to understand.

Q6: Can you please write JUnit test for the code?

Sure, here is a sample JUnit test using the refactored code:

```
import org.junit.Assert;
import org.junit.Test;

public class SmartHomeTest {
    @Test
    public void testSmartHomeController() {
        Device fan = new Fan();
        Device light = new Light();
        Control smartHomeController = new SmartHomeController();
        smartHomeController.setDevice(fan);
        smartHomeController.turnOn();
        Assert.assertEquals(State.ON, smartHomeController.getState());
        smartHomeController.turnOff();
        Assert.assertEquals(State.OFF, smartHomeController.getState());
        smartHomeController.setDevice(light);
        smartHomeController.turnOn();
        Assert.assertEquals(State.ON, smartHomeController.getState());
        smartHomeController.turnOff();
        Assert.assertEquals(State.OFF, smartHomeController.getState());
    }
}
```

In this test, we create a Fan and Light device, a SmartHomeController object, and set the device of the controller to the fan. We then turn the fan on and check if the state is ON. We then turn the fan off and check that the state is OFF. We then set the device of the controller to the light and repeat the same process.

Q7: What is the test coverage of this test? [give test code on Q6]

This code is testing the SmartHomeController class using JUnit. The test creates an instance of a Fan and a Light object, creates an instance of the SmartHomeController class, sets the Fan as the device to be controlled, turns the device on, and then checks that the state is ON. It then turns the device off and checks that the state is OFF. The test then does the same with the Light object. In terms of test coverage, this code is testing the functionality of the SmartHomeController class and ensuring that it is working as expected. However, it does not provide any information about the overall code coverage of the project.