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. [Repeat Q4 with code]
6. Does this code still use the mediator pattern?
7. Does this code have code smells? [give the refactor code]
8. Can you please write JUnit test for the code?
9. What is the test coverage of the JUnit test?

Replit Ghostwriter answers

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

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

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 followCommand(String command, Appliance appliance);
}

class SmartHomeControl implements Control {

    private List<Appliance> appliances = new ArrayList<Appliance>();

    public void followCommand(String command, Appliance appliance) {
        if(command.equalsIgnoreCase("TURNON")) {
            appliance.turnOn();
        }
        else if(command.equalsIgnoreCase("TURNOFF")) {
            appliance.turnOff();
        }
        appliances.add(appliance);
    }
}
```

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

Sure thing! Here are some code smells that I see:

The Appliance interface has methods that are specific to turning an appliance on and off, but maybe not all appliances have that functionality or use the same mechanism to achieve it. That could lead to issues if someone were to implement the interface for a device that doesn't support these actions or uses different names for them.

Since both Fan and Light classes have similar properties, it could be a good idea to create a superclass to avoid duplication of code.

The SmartHomeControl class adds the appliance to a list every time a command is executed. This may not be necessary if there is no future use of the appliances list.

The control commands are passed as string parameters, which could lead to a typo or spelling errors. A better approach would be to declare an enum for the available commands.

The method names of turnOn() and turnOff() use different cases. It would be best to keep the same convention.

These are just some suggestions for improvement to make the code more readable and maintainable.

Q3: Can you help me refactor the code?

Sure! Here are some changes we can make to address the code smells:

Instead of having an appliance interface with the turnOn() and turnOff() methods, we can use an abstract class where these methods are implemented and any appliance that inherits from this parent class will get these features.

We can create an ApplianceType enum to avoid spelling errors.

We can create an ApplianceFactory class to handle the creation of appliances in a centralized place.

Here's the refactored code:

```
abstract class Appliance {
    protected String state;
    public abstract void togglePower();

    public String getState() {
        return this.state;
    }
}

class Fan extends Appliance {

    public void togglePower() {
        if (state == null || state.equalsIgnoreCase("OFF")) {
            this.state = "ON";
            System.out.println("Fan turned ON");
        } else {
            this.state = "OFF";
            System.out.println("Fan turned OFF");
        }
    }
}

class Light extends Appliance {

    public void togglePower() {
        if (state == null || state.equalsIgnoreCase("OFF")) {
            this.state = "ON";
            System.out.println("Light turned ON");
        } else {
            this.state = "OFF";
            System.out.println("Light turned OFF");
        }
    }
}

enum ApplianceType {
    FAN,
    LIGHT
}

class ApplianceFactory {
    public static Appliance createAppliance(ApplianceType type) {
        switch (type) {
            case FAN:
                return new Fan();
            case LIGHT:
                return new Light();
            default:
                return null;
        }
    }
}

class SmartHomeControl {

    public void followCommand(ApplianceType type, String command) {
        Appliance appliance = ApplianceFactory.createAppliance(type);
        if (appliance == null) {
            System.out.println("Appliance not supported");
            return;
        }

        if(command.equalsIgnoreCase("TOGGLE")) {
            appliance.togglePower();
        }
    }
}
```

Q4: Does the refactored code still use the mediator pattern?

I'm sorry, but I would need more context to be able to answer this question. Can you provide more information about the code you are referring to and which specific part has been refactored?

Q5: [Repeat Q4 with code]

```
abstract class Appliance {
    protected String state;

    public abstract void togglePower();
    public String getState() {
        return this.state;
    }
}

class Fan extends Appliance {

    public void togglePower() {
        if (state == null || state.equalsIgnoreCase("OFF")) {
            this.state = "ON";
            System.out.println("Fan turned ON");
        } else {
            this.state = "OFF";
            System.out.println("Fan turned OFF");
        }
    }
}

class Light extends Appliance {

    public void togglePower() {
        if (state == null || state.equalsIgnoreCase("OFF")) {
            this.state = "ON";
            System.out.println("Light turned ON");
        } else {
            this.state = "OFF";
            System.out.println("Light turned OFF");
        }
    }
}

enum ApplianceType {
    FAN,
    LIGHT
}

class ApplianceFactory {
    public static Appliance createAppliance(ApplianceType type) {
        switch (type) {
            case FAN:
                return new Fan();
            case LIGHT:
                return new Light();
            default:
                return null;
        }
    }
}

class SmartHomeControl {

    public void followCommand(ApplianceType type, String command) {
        Appliance appliance = ApplianceFactory.createAppliance(type);
        if (appliance == null) {
            System.out.println("Appliance not supported");
            return;
        }

        if(command.equalsIgnoreCase("TOGGLE")) {
            appliance.togglePower();
        }
    }
}
```

** Explain code**

This code defines an abstract class called Appliance that contains an instance variable state and two methods: togglePower() and getState(). Subclasses of Appliance are then defined (Fan and Light), which implement the togglePower() method in their own way. An enum called ApplianceType defines the types of appliances that can be created, and a class called ApplianceFactory is defined, which creates a new appliance of the specified type. Finally, a class called SmartHomeControl is defined, which receives a command and an appliance type as input, creates an appliance of the specified type, and toggles the state of the appliance or prints an error message if the appliance is not supported.

Q6: Does this code still use the mediator pattern?

Yes, this code still uses the mediator pattern.

The ApplianceFactory acts as the mediator here as it abstracts the creation of concrete objects from the SmartHomeControl. Once an Appliance is created, SmartHomeControl passes on a command to the created Appliance through the togglePower() method.

This makes it easy to add new appliance types in the future as the SmartHomeControl does not need to be updated when a new appliance is added. The SmartHomeControl updates the ApplianceType enum to accommodate it, and the ApplianceFactory handles the instantiation of Appliance objects.

Q7: Does this code have code smells?

This code defines a few classes and an enum. The Appliance is an abstract class that has a method to toggle power and returns the current state of the appliance. The Fan and Light classes inherit from Appliance and implement the togglePower method to turn the appliance on or off. The ApplianceFactory class is a factory pattern implementation that creates Appliance objects depending on the ApplianceType passed to the createAppliance method. Finally, the SmartHomeControl allows users to create Appliance objects and toggle their state through string commands.

As for code smells, there are a few:

- state in the Appliance class is not initialized, which can cause NullPointerExceptions if it is not given a value in any of the subclasses. Instead of this, you can have a stateless base class and make the state a field in the Fan and Light classes.
- The createAppliance method of the ApplianceFactory class returns null in the default block of the switch statement if an unsupported ApplianceType is passed. It would be better to throw an IllegalArgumentException or a custom UnsupportedApplianceException to provide a clear and scalable error message.
- The SmartHomeControl class doesn't allow for additional parameters beyond toggling the appliance on or off. It may be worth considering additional command string formatting or creating separate command classes for more extensive control.

Q8: Can you please write JUnit test for the code?

Sure! Here is an example of JUnit tests for the Appliance classes.

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

public class ApplianceTest {

    @Test
    public void testFanTurnsOn() {
        Fan fan = new Fan();
        fan.togglePower();
        String expected = "ON";
        String actual = fan.getState();
        assertEquals(expected, actual);
    }

    @Test
    public void testFanTurnsOff() {
        Fan fan = new Fan();
        fan.togglePower();
        fan.togglePower();
        String expected = "OFF";
        String actual = fan.getState();
        assertEquals(expected, actual);
    }

    @Test
    public void testLightTurnsOn() {
        Light light = new Light();
        light.togglePower();
        String expected = "ON";
        String actual = light.getState();
        assertEquals(expected, actual);
    }

    @Test
    public void testLightTurnsOff() {
        Light light = new Light();
        light.togglePower();
        light.togglePower();
        String expected = "OFF";
        String actual = light.getState();
        assertEquals(expected, actual);
    }

    @Test(expected=IllegalArgumentException.class)
    public void testCreateUnsupportedAppliance() {
        ApplianceFactory.createAppliance(null);
    }
}
```

These tests ensure that each Appliance object can turn on and off, with the correct state being retained after each. The last test case checks for unsupported appliance type and confirms that it throws the appropriate exception.

Q9: What is the test coverage of the JUnit test?

I cannot determine the code coverage of the JUnit tests without additional information. The code coverage of the JUnit tests depends on the amount of code executed during the tests. To find the code coverage of the JUnit tests, you can use a code coverage tool that measures which parts of the code were executed during the test run. There are many code coverage tools for Java, like JaCoCo or Cobertura, that can work with JUnit tests and give you the code coverage percentage.

Q10: What is the test coverage of this test? [give test code on Q8]

This code defines five test methods for the Appliance class. Specifically:

The first two test methods test that a fan can be turned on and off correctly.
The next two test methods test that a light can be turned on and off correctly.
The final test method tests that the createAppliance method of ApplianceFactory throws an IllegalArgumentException when passed a null argument.
There is no visible information in the code regarding test coverage (number of lines covered by test cases).