JAVA -Junit

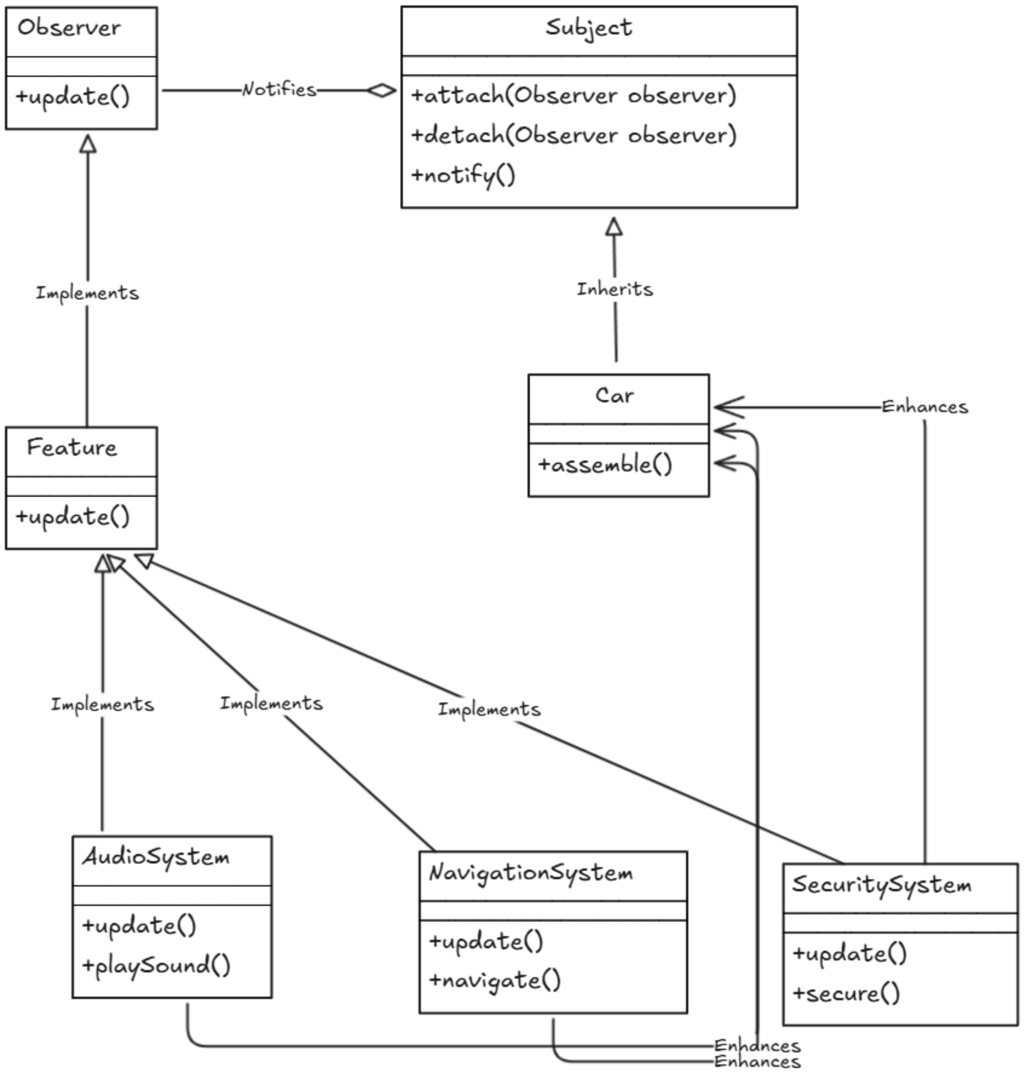
Prompt

-Observer

Use the Observer pattern to write code in Java programming language and write JUnit to test the given code that have 100% statement coverage from the requirements below. Use Observer pattern to design automotive system development by adding features to cars, such as audio system, navigation system or security systems without changing the main structure of the car.

Diagram

Observer



Observer

รอบที่1

AudioSystem

```
// Concrete Observer (ConcreteObserver) - AudioSystem
class AudioSystem implements CarObserver {
    @Override
    public void update(String feature) {
        if (feature.equalsIgnoreCase("Audio System")) {
            System.out.println("Audio System installed and ready.");
        }
    }
}
```

AutomotiveSystemTest

```
// AutomotiveSystemTest.java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class AutomotiveSystemTest {

    @Test
    public void testObserverPattern() {
        Subject car = new Subject();

        AudioSystem audio = new AudioSystem();
        NavigationSystem navigation = new NavigationSystem();
        SecuritySystem security = new SecuritySystem();

        // Add observers
        car.addObserver(audio);
        car.addObserver(navigation);
        car.addObserver(security);

        // Test initial update
        car.setFeatureUpdate("Bluetooth connectivity");
        assertEquals("Audio System updated with: Bluetooth connectivity", audio.getFeatureUpdate());
        assertEquals("Navigation System updated with: Bluetooth connectivity",
navigation.getFeatureUpdate());
        assertEquals("Security System updated with: Bluetooth connectivity", security.getFeatureUpdate());

        // Remove an observer and update again
        car.removeObserver(security);
        car.setFeatureUpdate("GPS tracking");
        assertEquals("Audio System updated with: GPS tracking", audio.getFeatureUpdate());
        assertEquals("Navigation System updated with: GPS tracking", navigation.getFeatureUpdate());
        assertNull(security.getFeatureUpdate(), "Security System should not receive updates after being
removed");
    }
}
```

BasicCar

```
// Concrete Observable (ConcreteSubject)
class BasicCar implements Car {
    private List<CarObserver> observers = new ArrayList<>();

    @Override
    public void attach(CarObserver observer) {
        observers.add(observer);
    }

    @Override
    public void detach(CarObserver observer) {
        observers.remove(observer);
    }


    @Override
    public void notifyObservers(String feature) {
        for (CarObserver observer : observers) {
            observer.update(feature);
        }
    }


    public void addFeature(String feature) {
        System.out.println("Adding feature: " + feature);
        notifyObservers(feature);
    }
}
```

NavigationSystem

```
// Concrete Observer (ConcreteObserver) - NavigationSystem
class NavigationSystem implements CarObserver {
    @Override
    public void update(String feature) {
        if (feature.equalsIgnoreCase("Navigation System")) {
            System.out.println("Navigation System installed and ready.");
        }
    }
}
```

Observer

```
// Observer interface
interface CarObserver {
    void update(String feature);
}
```

SecuritySystem

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CarFeatureTest {

    @Test
    void testCarFeatureAddition() {
        // Create a basic car
        BasicCar car = new BasicCar();

        // Create observers
        AudioSystem audioSystem = new AudioSystem();
        NavigationSystem navigationSystem = new NavigationSystem();
        SecuritySystem securitySystem = new SecuritySystem();

        // Attach observers to the car
        car.attach(audioSystem);
        car.attach(navigationSystem);
        car.attach(securitySystem);

        // Add features to the car
        car.addFeature("Audio System");
        car.addFeature("Navigation System");
        car.addFeature("Security System");

        // Assertions to check if features are installed (not directly testable here,
        // but you'll see the output in the console).
        assertTrue(true); // Placeholder assertion, the real test is the console output.
    }
}
```

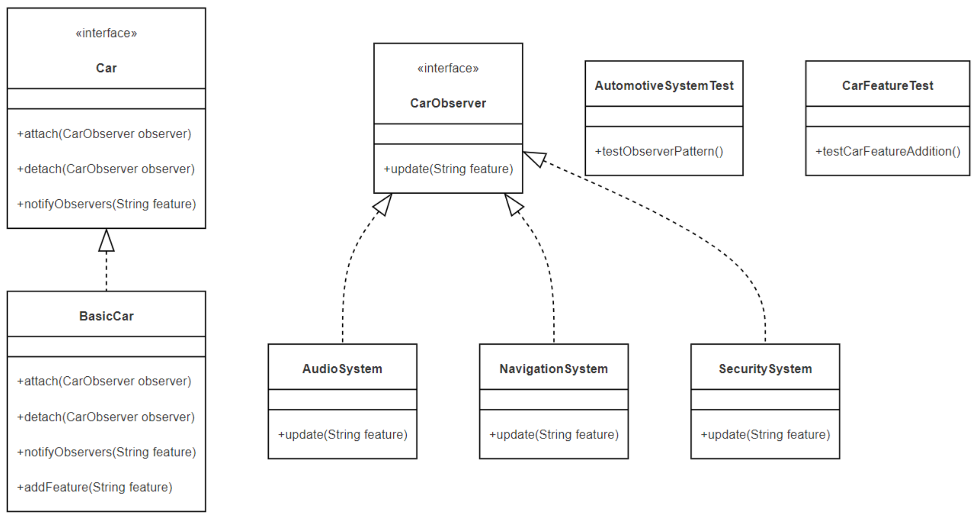
SecuritySystemDecorator

```
// Concrete Decorator class for Security System
class SecuritySystem extends CarDecorator {
    public SecuritySystem(Car decoratedCar) {
        super(decoratedCar);
    }

    @Override
    public String getDescription() {
        return super.getDescription() + ", Security System";
    }

    @Override
    public double getCost() {
        return super.getCost() + 800;
    }
}
```

Subject

```
// Observable interface (Subject)
interface Car {
    void attach(CarObserver observer);
    void detach(CarObserver observer);
    void notifyObservers(String feature);
}
```

Diagram From Code:



โค้ดที่แก้ไข:

```
1.AudioSystem.java(add)
private String featureUpdate;
public String getFeatureUpdate() {
        return featureUpdate;
    }

2.NavigationSystem.java(add)
private String featureUpdate;
public String getFeatureUpdate() {
        return featureUpdate;
    }

3.SecuritySystem.java(add)
private String featureUpdate;
public String getFeatureUpdate() {
        return featureUpdate;
    }

4.BasicCar.java (add)
private List<CarObserver> observers = new ArrayList<>(); --> private final List<CarObserver> observers =
new ArrayList<>();

5.Subject.java (add)
    private final List<CarObserver> observers = new ArrayList<>();
    private String featureUpdate;
    // Method to add an observer
    public void addObserver(CarObserver observer) {
        observers.add(observer);
    }
    // Method to remove an observer
    public void removeObserver(CarObserver observer) {
        observers.remove(observer);
    }
    // Method to set feature update and notify all observers
    public void setFeatureUpdate(String featureUpdate) {
        this.featureUpdate = featureUpdate;
        notifyAllObservers();
    }
    // Method to notify all observers of a feature update
    private void notifyAllObservers() {
        for (CarObserver observer : observers) {
            observer.update(featureUpdate);
        }
    }
}
```