

Applying the Observer Design Pattern to Sensor Data Monitoring in Self-Driving Cars

Frida Cano Falcón

Java Backend Academy MTY
August 2024

Week 3

August 30, 2024

Introduction

The Observer design pattern is a behavioral pattern that provides a way to notify multiple objects about changes in the state of a subject (or observable). This pattern is particularly useful in scenarios where one object needs to update multiple dependent objects without tightly coupling them. In the context of a self-driving car, where various sensors and systems need to react to real-time changes in sensor data, the Observer pattern is a powerful tool to ensure that all components stay in sync with the latest data.

This report demonstrates the implementation of the Observer design pattern in a sensor data monitoring system for a self-driving car. The focus is on managing and reacting to data from different sensors, such as cameras, LiDAR, and radar, and ensuring that various systems within the car (like obstacle detection, lane tracking, and traffic sign recognition) can stay updated with real-time data.

Project Overview

1. Subject Interface

The *Subject* interface defines the methods required to attach, detach, and notify observers. It is implemented by the sensor classes to manage and notify their observers.

```
src > com > autonomouscar > sensors > J Subject.java >
1  package com.autonomouscar.sensors;
2
3  public interface Subject {
4      void attach(Observer observer);
5      void detach(Observer observer);
6      void notifyObservers();
7  }
```

2. Observer Interface

The *Observer* interface declares the *update* method, which is called by the subject to notify the observer of any changes.

```
src > com > autonomouscar > sensors > J Observer.java > ...  
1  package com.autonomouscar.sensors;  
2  
3  public interface Observer {  
4      void update(String data);  
5  }
```

3. Concrete Subject Classes

CameraSensor.java

The *CameraSensor* class is a concrete implementation of the *Subject* interface. It manages a list of observers and notifies them with new data.

```

src > com > autonomouscar > sensors > J CameraSensor.java
1  package com.autonomouscar.sensors;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class CameraSensor implements Subject {
7
8      private List<Observer> observers = new ArrayList<>();
9      private String data;
10
11     @Override
12     public void attach(Observer observer) {
13         observers.add(observer);
14     }
15
16     @Override
17     public void detach(Observer observer) {
18         observers.remove(observer);
19     }
20
21     @Override
22     public void notifyObservers() {
23         for (Observer observer : observers) {
24             observer.update(data);
25         }
26     }
27
28     public void setData(String data) {
29         this.data = data;
30         notifyObservers();
31     }
32 }
33

```

LiDARSensor.java

The *LiDARSensor* class is another concrete implementation of the *Subject* interface, similar to *CameraSensor*.

```

src > com > autonomouscar > sensors > J LiDARSensor.java > ...
1  package com.autonomouscar.sensors;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class LiDARSensor implements Subject {
7
8      private List<Observer> observers = new ArrayList<>();
9      private String data;
10
11     @Override
12     public void attach(Observer observer) {
13         observers.add(observer);
14     }
15
16     @Override
17     public void detach(Observer observer) {
18         observers.remove(observer);
19     }
20
21     @Override
22     public void notifyObservers() {
23         for (Observer observer : observers) {
24             observer.update(data);
25         }
26     }
27
28     public void setData(String data) {
29         this.data = data;
30         notifyObservers();
31     }
32 }

```

4. Concrete Observer Classes

ObstacleDetectionSystem.java

The *ObstacleDetectionSystem* class implements the *Observer* interface and processes data updates for obstacle detection.

```
src > com > autonomouscar > sensors > J ObstacleDetectionSystem.java > ...
1  package com.autonomouscar.sensors;
2
3  public class ObstacleDetectionSystem implements Observer {
4
5      @Override
6      public void update(String data) {
7          System.out.println("Obstacle Detection System received data: " + data);
8          // Process the data to detect obstacles
9      }
10 }
```

LaneTrackingSystem.java

The *LaneTrackingSystem* class implements the *Observer* interface and processes data updates for lane tracking.

```
src > com > autonomouscar > sensors > J LaneTrackingSystem.java > ...
1  package com.autonomouscar.sensors;
2
3  public class LaneTrackingSystem implements Observer {
4
5      @Override
6      public void update(String data) {
7          System.out.println("Lane Tracking System received data: " + data);
8          // Process the data to track lanes
9      }
10 }
```

TrafficSignRecognitionSystem.java

The *TrafficSignRecognitionSystem* class implements the *Observer* interface and processes data updates for traffic sign recognition.

```
src > com > autonomouscar > sensors > J TrafficSignRecognitionSystem.java > ...
1  package com.autonomouscar.sensors;
2
3  public class TrafficSignRecognitionSystem implements Observer {
4
5      @Override
6      public void update(String data) {
7          System.out.println("Traffic Sign Recognition System received data: " + data);
8          // Process the data to recognize traffic signs
9      }
10 }
```

5. Main Class

The *VisionSystem* class demonstrates how the Observer pattern is used to manage and react to sensor data updates.

```
src > com > autonomouscar > sensors > VisionSystem.java
1  package com.autonomouscar.sensors;
2
3  public class VisionSystem {
    Run | Debug
4      public static void main(String[] args) {
5          System.out.println(x:"Initializing Vision System");
6          // Create sensor subjects
7          CameraSensor cameraSensor = new CameraSensor();
8          LiDARSensor lidarSensor = new LiDARSensor();
9
10         // Create observer systems
11         ObstacleDetectionSystem obstacleDetection = new ObstacleDetectionSystem();
12         LaneTrackingSystem laneTracking = new LaneTrackingSystem();
13         TrafficSignRecognitionSystem trafficSignRecognition = new TrafficSignRecognitionSystem();
14
15         // Attach observers to sensors
16         cameraSensor.attach(obstacleDetection);
17         cameraSensor.attach(laneTracking);
18         lidarSensor.attach(obstacleDetection);
19         lidarSensor.attach(trafficSignRecognition);
20
21         // Simulate sensor data updates
22         cameraSensor.setData(data:"New camera image data");
23         lidarSensor.setData(data:"New LiDAR depth data");
24     }
25 }
```

Conclusion

The Observer design pattern has been effectively applied to manage and react to sensor data in a self-driving car's perception system. By implementing this pattern, the system ensures that various modules (like obstacle detection, lane tracking, and traffic sign recognition) remain updated with real-time sensor data. This approach promotes loose coupling between sensor data producers and consumers, allowing for scalable and maintainable code. The Observer pattern enhances the system's ability to react to changes in sensor data dynamically, which is crucial for the reliable operation of autonomous vehicles.