Progetto in Java per la gestione di un'auto a guida autonoma

Elisa De Filomeno

A.A. 2020-2021

Contenuti

1 Introduzione

L'elaborato consiste nella realizzazione di un progetto di microprogettazione scritto nel linguaggio Java. Esso simula, in modo semplificato, il meccanismo di funzionamento della guida autonoma di un veicolo. L'obiettivo è quello di fornire una struttura adeguata per problemi di gestione analoghi, mostrando il modo con cui le varie classi si relazionano tra loro attraverso la combinazione dei design pattern Factory Method, Iterator, Observer e Builder. Seguendo l'impianto fornito si favorisce una possibile realizzazione di implementazioni più complesse delle singole funzioni o l'aggiunta di ulteriori classi.

2 Design Patterns

2.1 Observer

Si tratta di un pattern comportamentale che viene utilizzato quando si vuole realizzare una dipendenza uno-a-molti, in cui il cambiamento di stato di un soggetto (Subject) viene notificato a tutti oggetti (Observer) interessati.

2.1.1 Struttura

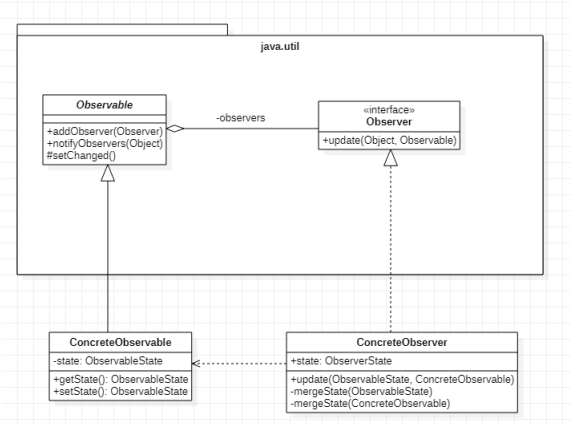

Figure 1: Observer Class Diagram

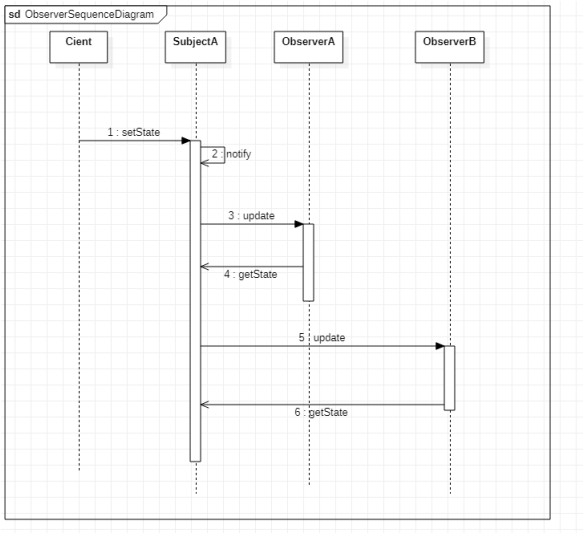

Figure 2: Observer pull mode Sequence Diagram

2.1.2 Partecipanti

Questo pattern è composto dai seguenti partecipanti:

- Subject: espone l'interfaccia che consente agli osservatori di iscriversi e cancellarsi; mantiene una reference a tutti gli osservatori iscritti
- Observer: espone l'interfaccia che consente di aggiornare gli osservatori in caso di cambio di stato del soggetto osservato.
- ConcreteSubject: mantiene lo stato del soggetto osservato e notifica gli osservatori in caso di un cambio di stato.
- ConcreteObserver: implementa l'interfaccia dell'Observer definendo il comportamento in caso di cambio di stato del soggetto osservato

2.1.3 Conseguenze

Tale pattern presenta i seguenti vantaggi/svantaggi:

- Astratto accoppiamento tra Subject e Observer: il Subject sa che una lista di Observer sono interessati al suo stato ma non conosce le classi concrete degli Observer, pertanto non vi è un accoppiamento forte tra di loro.
- Notifica diffusa: il Subject deve notificare a tutti gli Observer il proprio cambio di stato, gli Observer sono responsabili di aggiungersi e rimuoversi dalla lista.
- Lo svantaggio è che il codice risulta più complesso e più difficilmente modificabile.

2.2 Iterator

L'Iterator è un pattern comportamentale basato su oggetti e viene utilizzato quando, dato un aggregato di oggetti, si vuole accedere ai suoi elementi senza dover esporre la sua struttura. L'obiettivo di questo pattern è quello di disaccoppiare l'utilizzatore e l'implementatore dell'aggregazione di dati, tramite un oggetto intermedio che esponga sempre gli stessi metodi indipendentemente dall'aggregato di dati.

2.2.1 Struttura

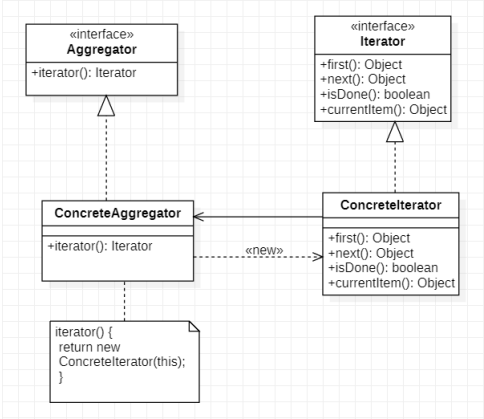

Figure 3: Iterator Class Diagram

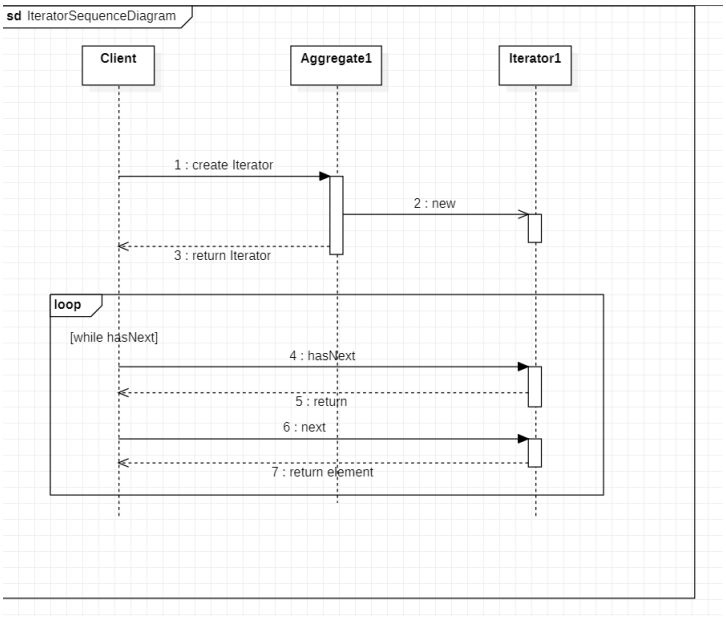

Figure 4: Iterator Sequence Diagram

2.2.2 Partecipanti

Questo pattern è composto dai seguenti partecipanti:

- Iterator: colui che espone i metodi di accesso alla struttura dati
- ConcreteIterator: implementa l'Iteratore e tiene il puntatore alla struttura dati
- Aggregator: definisce l'interfaccia per creare un oggetto di tipo Iteratore
- ConcreteAggregator: implementa l'interfaccia di creazione di un oggetto Iteratore

2.2.3 Conseguenze

Tale pattern presenta i seguenti vantaggi/svantaggi:

- unica interfaccia di accesso: l'accesso ai dati avviene tramite l'Iterator che espone un'unica interfaccia e nasconde le diverse implementazioni degli Aggregator
- diversi iteratori di accesso: l'Aggregator può essere attraversato tramite diversi Iterator in cui ogni Iterator nasconde un algoritmo diverso

2.3 Builder con Static Factory Method

Il Builder design pattern è un pattern creazionale che permette di costruire oggetti complessi in modo facilitato, passo-passo. La responsabilità della creazione è delegata a una classe Builder, in genere una static inner class della classe di cui si vogliono costruire oggetti. Essa si bassa sulla fluent interface. I metodi factory statici permettono di incapsulare la creazione di oggetti, tuttavia non sono un design pattern perché non astraggano dal tipo concreto di oggetto creato. Se presenti nella stessa classe che si vuole istanziare possono rappresentare un'alternativa ai costruttori, i quali diventano privati. Il vantaggio dei metodi factory statici è di rendere il codice più leggibile, in quanto possono avere nomi più significativi rispetto ai costruttori. Inoltre, essendo il costruttore privato, una sua futura modifica nella signature non va a "rompere" i client esistenti, basta modificare solo il codice della classe.

2.3.1 Struttura

Ci sono diverse forme del Builde design pattern. Riporto un class Diagram più simile a quello utilizzato nel progetto.

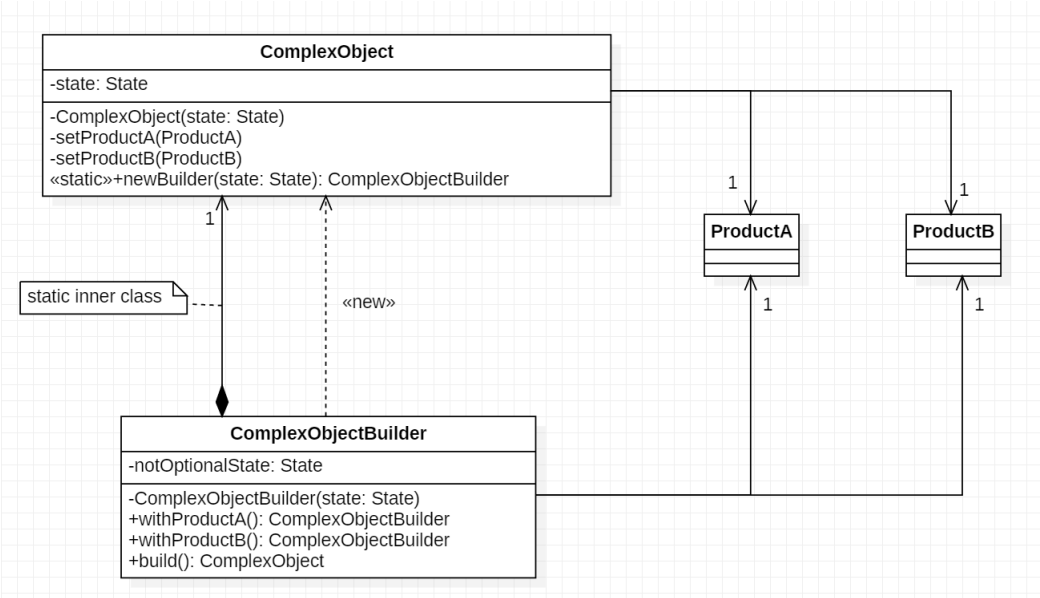

Figure 5: Builder with static Factory Method Class Diagram

2.3.2 Partecipanti

- ComplexObject: oggetto complesso che viene costruito dal ComplexObjectBuilder.

- ComplexObjectBuilder: costruisce e assembla le parti dell'oggetto complesso e tiene traccia della rappresentazione che crea.

2.3.3 Conseguenze

Tale pattern presenta i seguenti vantaggi/svantaggi:

- La costruzione di un oggetto complesso e la sua rappresentazione vengono isolate.
- La creazione dell'oggetto complesso avviene in modo più semplice, leggibile e controllato.
- Lo svantaggio è che il codice risulta più complesso e più difficilmente modificabile.

3 Progetto

4 Analisi dei Requisiti

4.1 Casi d'uso

L'applicazione espone diverse funzioni che sono rese disponibili ad un Utente. Queste funzioni permettono di costruire nel programma un veicolo e attivargli la modalità di guida autonoma. La funzione di guida autonoma sottende l'inserimento della posizione corrente del veicolo e della posizione di destinazione. L'Utente può inoltre gestire vari accessori elettrici del veicolo, quali il finestrino elettrico, il tergicristallo e lo specchietto retrovisore.

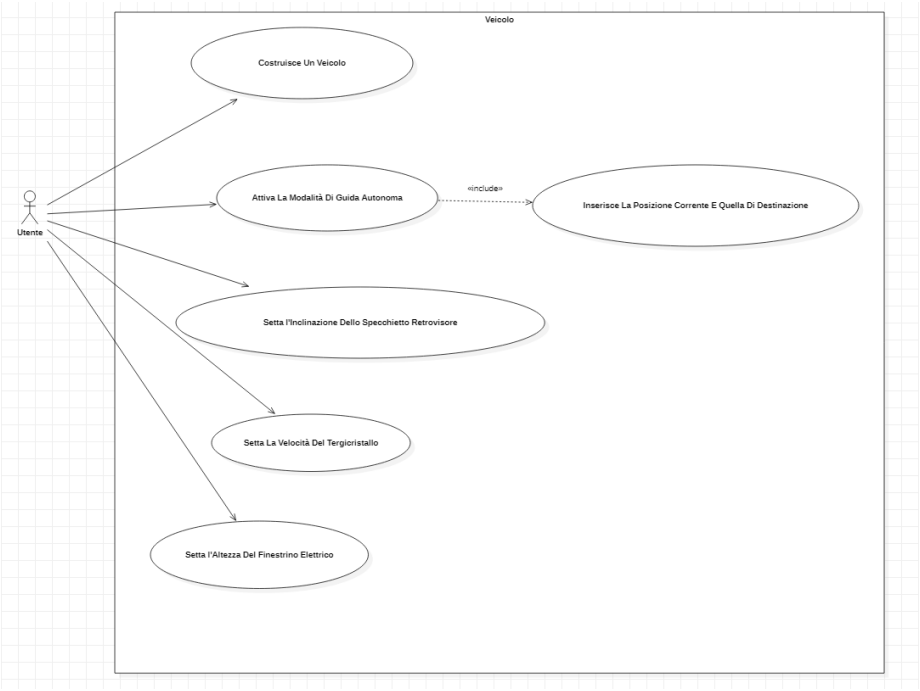

Figure 6: Use Case Diagram

5 Progettazione

Il progetto è strutturato dividendo il veicolo in sensori, attuatori e componenti che controllano gli attuatori rielaborando i dati forniti dai sensori. I sensori si occupano di estrarre informazioni dal mondo circostante. Questi dati vengono poi passati, per essere rielaborati, alle componenti intelligenti del sistema che impartiscono comandi agli attuatori. Gli attuatori compiono azioni sull'ambiente, grazie a quest'ultimi è possibile ad esempio manovrare il volante del veicolo per curvare o si possono fare azioni più ordinarie come abbassare il finestrino elettrico.

5.1 Metodo

Per la realizzazione è stato utilizzato il linguaggio di programmazione Java attraverso Eclipse. Sono state seguite vari fasi di sviluppo: nella fase di analisi dei requisiti sono stati identificati i casi d'uso e rappresentati attraverso gli Use Case Diagrams. Nella struttura principale del programma sono stati usati i pattern Builder, Iterator e Observer.

5.2 Class Diagram

Il Diagramma delle Classi in UML rappresenta la progettazione delle classi, esso mostra come vengono partizionate le responsabilità tra le classi. Il Veicolo ha più Sensori e più Attuatori, fra cui gli attuatori per la guida autonoma (Attuatore del Freno, Acceleratore e Volante), e i Controllori per la gestione della guida autonoma e la gestione degli accessori. La Cpu tiene le collezioni dei Sensori e degli Attuatori del veicolo, oltre a un riferimento al Sistema di Navigazione. Essa genera Protocolli per il controllo della guida autonoma. Il Protocollo è un insieme di Istruzioni sequenzializzate che gli Attuatori per la guida autonoma eseguono. L'ambiente circostante al veicolo è rappresentato come un Mappa 3D che contiene un insieme di Posizioni. Il Sistema di Navigazione contiene la Posizione corrente e quella di destinazione del veicolo.

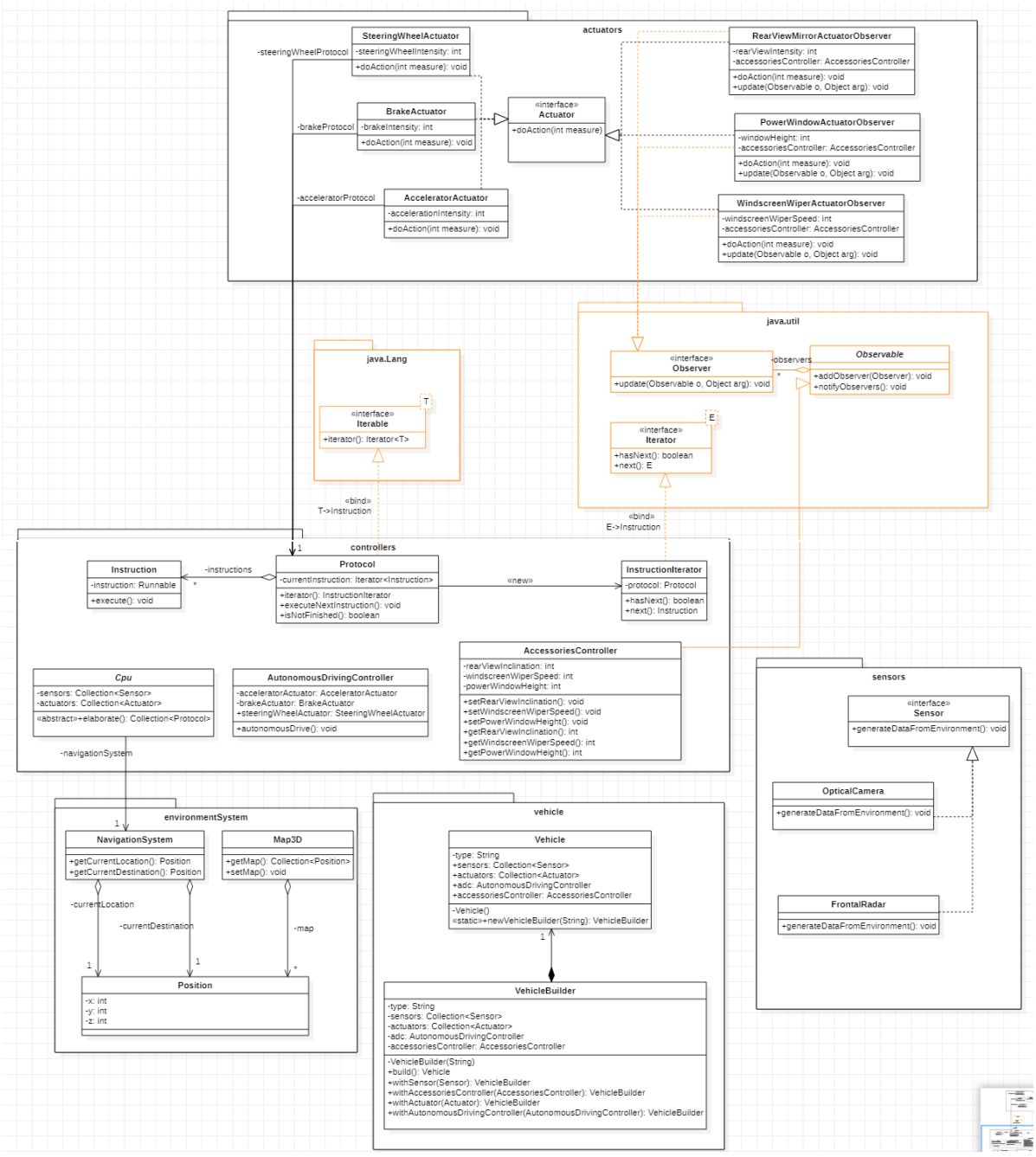

Figure 7: Class Diagram

6 Implementazione

Riporto l'organizzazione del codice in packages:

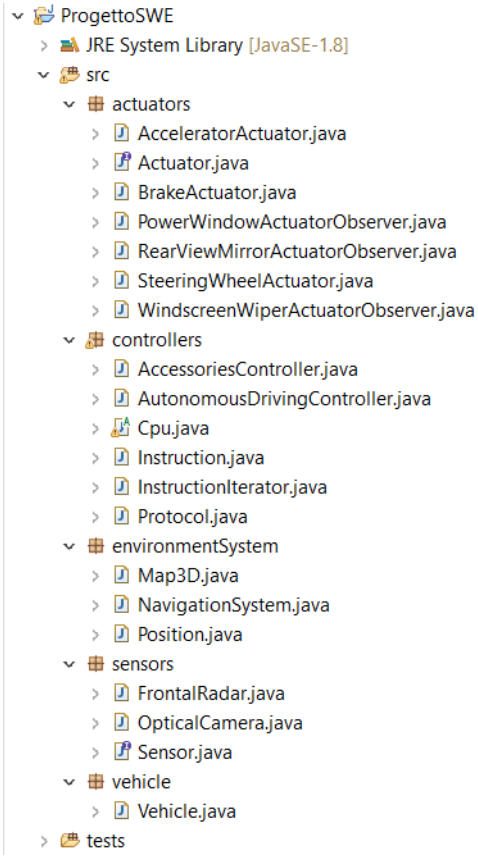

Figure 8: Organizzazione del codice in packages

Per l'implementazione sono state definite nuove classi ed interfacce o riutilizzate quelle della libreria standard (e.g. le interfacce contenute nel package java.util come Iterator⟨E⟩ e Observer).

- Instruction: ha un riferimento a una variabile di tipo Runnable (interfaccia funzionale) e il suo unico metodo execute() invoca il metodo run() della variabile d'istanza.

```java
public class Instruction{
    private Runnable instruction;


    public Instruction(Runnable instruction) {
        this.instruction = instruction;
    }

    public void execute() {
        this.instruction.run();
    }

}
```

Figure 9: Classe Instruction

- InstructionIterator: implementa Iterator⟨Instruction⟩, svolge il ruolo di ConcreteIterator nel design pattern Iterator. Implementa dunque i metodi next() e hasNext(). Contiene un riferimento a un Protocol, il ConcreteAggregator.

```java
public final class InstructionIterator implements Iterator<Instruction> {
    private Protocol protocol;


    public InstructionIterator(Protocol protocol) {
        super();
        this.protocol = protocol;
    }

    @Override
    public boolean hasNext() {
        return protocol.getInstructions().iterator().hasNext();
    }

    @Override
    public Instruction next() {
        return protocol.getInstructions().iterator().next();
    }
}
```

Figure 10: Classe InstructionIterator

- Protocol: implementa Iterable⟨Instruction⟩, svolge il ruolo di ConcreteAggregator nel design pattern Iterator. Implementa quindi il metodo iterator() che ritorna una nuova istanza della classe InstructionIterator(). Protocol contiene una collezione di Instruction e un riferimento all'Instruction corrente. Il metodo executeNextInstruction() esegue l'istruzione successiva a quella corrente.

```java
public class Protocol implements Iterable<Instruction>{
    private Collection<Instruction> instructions;
    private Iterator<Instruction> currentInstruction;

    public Protocol(Collection<Instruction> instructions) {
        this.instructions = instructions;
        currentInstruction= this.iterator();
    }

    public Collection<Instruction> getInstructions() {
        return instructions;
    }

    @Override
    public Iterator<Instruction> iterator() {
        return new InstructionIterator(this);
    }

    public void executeNextInstruction() {
        if(this.currentInstruction.hasNext()) {
            this.currentInstruction.next().execute();
            }
    }

    public boolean isNotFinished() {
        return currentInstruction.hasNext();
    }

}
```

Figure 11: Classe Protocol

- AutonomousDrivingController: la sua responsabilità è di controllare, attraverso l'esecuzione di protocolli, gli attuatori del freno, volante e acceleratore per la guida autonoma del veicolo. Tiene infatti nello stato i riferimenti a questi attuatori. Il metodo autonomousDrive() esegue tutte le istruzioni dei protocolli precedentemente elaborati dalla classe Cpu.

```java
public class AutonomousDrivingController
{
    private AcceleratorActuator acceleratorActuator;
    private BrakeActuator brakeActuator;
    private SteeringWheelActuator steeringWheelActuator;


    public AutonomousDrivingController(AcceleratorActuator acceleratorActuator, BrakeActuator brakeActuator,
            SteeringWheelActuator steeringWheelActuator) {
        this.acceleratorActuator = acceleratorActuator;
        this.brakeActuator = brakeActuator;
        this.steeringWheelActuator = steeringWheelActuator;
    }

    public void autonomousDrive() {
        while(acceleratorActuator.getAcceleratorProtocol().isNotFinished()||
              brakeActuator.getBrakeProtocol().isNotFinished()||
              steeringWheelActuator.getSteeringWheelProtocol().isNotFinished()) {

        acceleratorActuator.getAcceleratorProtocol().executeNextInstruction();
        brakeActuator.getBrakeProtocol().executeNextInstruction();
        steeringWheelActuator.getSteeringWheelProtocol().executeNextInstruction();
        }
    }
```

Figure 12: Frammento di codice della classe AutonomousDrivingController

La classe astratta Cpu ha per stato i Sensori del veicolo, i suoi Attuatori e il Sistema di Navigazione. Il suo unico metodo astratto è elaborate() che restituisce un insieme di Protocolli.

```java
public abstract class Cpu {
    private Collection<Sensor> sensors;
    private Collection<Actuator> actuators;
    private NavigationSystem navigationSystem;

    public abstract Collection<Protocol> elaborate();

}
```

Figure 13: Classe astratta Cpu

- AccessoriesController: estende la classe Observable presente nel package java.util. Il suo stato contiene lo stato generale di tutti gli accessori della macchina, come l'altezza del finestrino elettrico e la velocità di movimento attuale del tergicristallo. Gli attuatori interessati (Observer) al cambiamento di stato di AccessoriesController sono PowerWindowActuatorObserver, RearViewMirrorActuatorObserver e WindscreenWiperActuatorObserver.

```java
public class AccessoriesController
extends Observable
{
    private int rearViewInclination;
    private int windscreenWiperSpeed;
    private int powerWindowHeight;


    public void setRearViewInclination(int rearViewInclination) {
        this.rearViewInclination = rearViewInclination;
        this.setChanged();
        notifyObservers();
    }

    public void setWindscreenWiperSpeed(int windscreenWiperSpeed) {
        this.windscreenWiperSpeed = windscreenWiperSpeed;
        this.setChanged();
        notifyObservers();
    }
    public void setCarWindowHeight(int carWindowHeight) {
        this.powerWindowHeight = carWindowHeight;
        this.setChanged();
        notifyObservers();
    }
```

Figure 14: Frammento di codice della classe AccessoriesController

- Sensor (interfaccia): questa interfaccia rappresenta un generico sensore possibilmente presente nel veicolo ed ha un unico metodo generateDataFromEnvironment(). I Concrete Sensor implementano il metodo, in particolare:
  - OpticalCamera acquisisce dati come il colore del semaforo,
  - FrontalRadar acquisisce dati per rilevare gli ostacoli presenti in strada

```java
public interface Sensor {
    public void generateDataFromEnvironment();
}
```

Figure 15: Classe Sensor

- Actuator (interfaccia): questa interfaccia rappresenta un generico attuatore possibilmente presente nel veicolo ed ha un unico metodo doAction(int measure). I Concrete Actuator implementano il metodo, in particolare:
  - BrakeActuator agisce sull'intensità del freno ,
  - AcceleratorActuator agisce sull'intensità dell'acceleratore,
  - SteeringWheelActuator agisce sull'inclinazione del volante,

– PowerWindowActuatorObserver agisce sull'altezza del finestrino elettrico,
– RearViewMirrorActuatorObserver agisce sull'inclinazione del finestrino retrovisore,
– WindscreenWiperActuatorObserver agisce sulla velocità del tergicristallo.

Ad invocare il metodo possono essere le classi Controller del veicolo, ovvero AutonomousDrivingController e AccessoriesController.

```
1 package actuators;
2
3 public interface Actuator {
4     public void doAction(int measure);
5
6 }
```

Figure 16: Frammento di codice interfaccia Actuator

Dei ConcreteActuator, quelli necessari per la guida automa (BrakeActuator, AcceleratorActuator e SteeringWheelActuator) hanno inoltre un riferimento a una variabile di tipo Protocol. Ciascun protocollo verrà eseguito dal metodo autonomousDrive() della classe AutonomousDrivingController.

```
public final class BrakeActuator
implements Actuator{

    private int brakeIntensity;
    private Protocol brakeProtocol;

    @Override
    public void doAction(int measure) {
        this.setBrakeIntensity(measure);
    }

    private void setBrakeIntensity(int brakeIntensity) {
        this.brakeIntensity = brakeIntensity;
    }

    public int getBrakeIntensity() {
        return brakeIntensity;
    }
```

Figure 17: Frammento di codice della classe BrakeActuator

I ConcreteActuator inerenti invece agli accessori elettrici del veicolo imple-

mentano inoltre l'interfaccia Observer, svolgendo il ruolo di ConcreteObserver nel design pattern Observer (con la classe AccessoriesController come Observable). Il metodo update() evoca doAction(int).

```
public final class PowerWindowActuatorObserver
implements Observer, Actuator {

    private AccessoriesController accessoriesController;
    private int windowHeight;


    public PowerWindowActuatorObserver(AccessoriesController bcm) {
        this.accessoriesController = bcm;
        accessoriesController.addObserver(this);
    }

    @Override
    public void doAction(int measure) {
        this.setWindowHeight(measure);

    }

    @Override
    public void update(Observable o, Object arg) {
        doAction(accessoriesController.getCarWindowHeight());
    }

    private void setWindowHeight(int windowHeight) {
        this.windowHeight = windowHeight;
    }

    public int getWindowHeight() {
        return windowHeight;
    }
}
```

Figure 18: Classe PowerWindowActuatorObserver

- Vehicle: è una classe complessa da istanziare utilizzando il design pattern Builder con la classe interna statica VehicleBuilder. Ha il costruttore e i metodi setter privati, ma accessibili a VehicleBuilder. Il costruttore prende come unico elemento una stringa per la tipologia di veicolo, gli altri attributi (collezione di Sensori, di Attuatori e un riferimento a un AutonomousDrivingController e a AccessoriesController) sono opzionali e assemblati attraverso i metodi with di VehicleBuilder. La classe statica interna VehicleBuilder ha lo stesso stato di Vehicle e un costruttore che prende sempre una stringa. I metodi with permettono di modificare lo stato interno di VehicleBuilder e alla fine, con il metodo build(), si andrà a creare una nuova istanza di Vehicle con lo stato analogo a quello di VehicleBuilder.

```java
public class Vehicle{
    private String type;
    private Collection<Sensor> sensors;
    private Collection<Actuator> actuators;
    private AutonomousDrivingController adc;
    private AccessoriesController accessoriesController;

    private Vehicle(String type) {}

    public static VehicleBuilder newVehicleBuilder(String type) {
        return new VehicleBuilder(type);
    }

    public static class VehicleBuilder{
        private String type;
        private Collection<Sensor> sensors;
        private Collection<Actuator> actuators;
        private AutonomousDrivingController adc;
        private AccessoriesController accessoriesController;

        private VehicleBuilder(String type) {
            this.type=type;
            this.sensors= new ArrayList<Sensor>();
            this.actuators= new ArrayList<Actuator>();
        }

        public VehicleBuilder withSensor(Sensor sensor) {
            this.sensors.add(sensor);
            return this;
        }

        public VehicleBuilder withAccessoriesController(AccessoriesController accessoriesController) {
            this.accessoriesController= accessoriesController;
            return this;
        }

        public VehicleBuilder withActuator(Actuator actuator) {
            this.actuators.add(actuator);
            return this;
        }

        public VehicleBuilder withAutonomousDrivingController(AutonomousDrivingController adc) {
            this.actuators.add(adc.getAcceleratorActuator());
            this.actuators.add(adc.getBrakeActuator());
            this.actuators.add(adc.getSteeringWheelActuator());
            this.adc=adc;
            return this;
        }

        public Vehicle build() {
            Vehicle vehicle = new Vehicle(type);
            vehicle.setSensors(sensors);
            vehicle.setActuators(actuators);
            vehicle.setAdc(adc);
            vehicle.setAccessoriesController(accessoriesController);
            return vehicle;
        }
    }

    private void setSensors(Collection<Sensor> sensors) {
        this.sensors = sensors;
    }

    private void setActuators(Collection<Actuator> actuators) {
        this.actuators = actuators;
    }

    private void setAdc(AutonomousDrivingController adc) {
        this.adc = adc;
    }

    private void setAccessoriesController(AccessoriesController accessoriesController) {
        this.accessoriesController = accessoriesController;
    }
```

Figure 19: Classi Vehicle e VehicleBuilder

- Position: contiene per stato le coordinate di un qualsiasi oggetto che ha un riferimento a questa classe.

```
public final class Position {
    private int x;
    private int y;
    private int z;

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public int getZ() {
        return z;
    }
```

Figure 20: Frammento di codice della classe final Position

La classe final Map3D contiene una collezione di Posizioni e fornisce i metodi getter e setter sulla collezione. La classe NavigationSystem contiene nello stato due variabili di tipo Position che rappresentano la posizione corrente del veicolo e quella di destinazione.

```
public final class Map3D {
    Collection<Position> map;

    public Collection<Position> getMap() {
        return map;
    }

    public void setMap(Collection<Position> map) {
        this.map = map;
    }
}
```

Figure 21: Frammento di codice della classe final Map3D

```java
public class NavigationSystem {
    private Position currentLocation;
    private Position currentDestination;

    public Position getCurrentLocation() {
        return currentLocation;
    }
    public Position getCurrentDestination() {
        return currentDestination;
    }
```

Figure 22: Frammento di codice della classe NavigationSystem

Dettagli di implementazione

il progetto è stato realizzato utilizzando Eclipse. Per l'unit testing invece è stato usato il framework JUnit 4.0.

7 Unit Testing

Nel progetto sono stati verificati porzioni di codice, come metodi o classi, attraverso test cases. Nei metodi dei test vengono verificate delle asserzioni elementari utilizzando per esempio "assertEquals","assertTrue" e "assertFalse" di JUnit. I test sono stati scelti in prospettiva strutturale, in modalità white-box. Le classi di Test realizzate sono:

1. VehicleTest: in questa classe di test si è testata la classe Vehicle e VehicleBuilder. In particolare si è verificato che la creazione di Vehicle attraverso il pattern Builder avvenisse correttamente. I metodi testati sono:
   - newVehicleBuilder(String type), metodo statico di Vehicle
   - withSensor(Sensor), withActuator(Actuator), withAccessoriesController(AccessoriesController), withAutonomousDrivingController(AutonomousDrivingController) di VehicleBuilder
   - build() di VehicleBuilder

```java
public class VehicleTest {

    @Test
    public void test() {
        AccessoriesController accessoriesController = new AccessoriesController();
        Sensor camera = new OpticalCamera();
        Vehicle auto = Vehicle.newVehicleBuilder("automobile")
                .withSensor(camera)
                .withSensor(new FrontalRadar())
                .withAccessoriesController(accessoriesController)
                .withActuator(new PowerWindowActuatorObserver(accessoriesController))
                .withAutonomousDrivingController
                (new AutonomousDrivingController(new AcceleratorActuator(), new BrakeActuator(), new SteeringWheelActuator()))
                .build();


        assertSame(accessoriesController, auto.getAccessoriesController());
        assertEquals("automobile", auto.getType());
        assertTrue(auto.getSensors().contains(camera));
    }
}
```

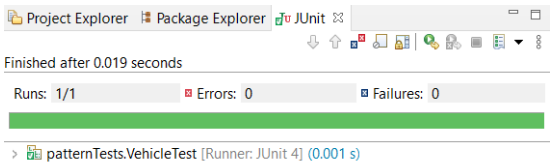

Figure 23: VehicleTest

2. AccessoriesControllerTest: in questa classe di test si sono testati i metodi setter della classe AccessoriesController (Observable) e in particolare si è verificato il giusto comportamento del pattern Observer.

```java
public class AccessoriesControllerTest {

    @Test
    public void test() {

        AccessoriesController bcm = new AccessoriesController();

        PowerWindowActuatorObserver WindowController = new PowerWindowActuatorObserver(bcm);
        RearViewMirrorActuatorObserver MirrorController = new RearViewMirrorActuatorObserver(bcm);
        WindscreenWiperActuatorObserver WiperController = new WindscreenWiperActuatorObserver(bcm);

        bcm.setCarWindowHeight(5);
        bcm.setRearViewInclination(12);
        bcm.setWindscreenWiperSpeed(17);

        assertEquals(5, WindowController.getWindowHeight());
        assertEquals(12, MirrorController.getRearViewInclination());
        assertEquals(17, WiperController.getWindscreenWiperSpeed());
    }
}
```

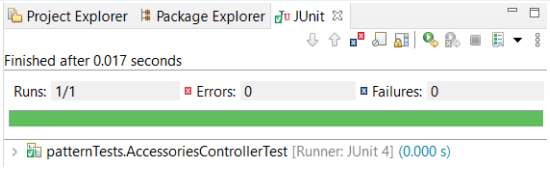

Figure 24: Accessories Controller Test

3. AutonomousDrivingControllerTest: in questa classe si test il metodo autonomousDrive() di AutonomousDrivingController. Vengono quindi creati i protocolli per il freno, acceleratore e volante e il metodo autonomousDrive() dovrà eseguirli.

```java
public class AutonomousDrivingControllerTest {

    @Test
    public void test() {
        AcceleratorActuator acceleratorActuator= new AcceleratorActuator();
        BrakeActuator brakeActuator= new BrakeActuator();
        SteeringWheelActuator steeringWheelActuator= new SteeringWheelActuator();
        AutonomousDrivingController adECU = new AutonomousDrivingController(acceleratorActuator, brakeActuator, steeringWheelActuator);


        Collection<Instruction> accelatorCommands_Uturn = new ArrayList<>();
        Collection<Instruction> steeringWheelCommands_Uturn = new ArrayList<>();
        Collection<Instruction> brakeCommands_Uturn = new ArrayList<>();


        accelatorCommands_Uturn.add(new Instruction(()->{acceleratorActuator.doAction(-3);}));
        accelatorCommands_Uturn.add(new Instruction(()->{acceleratorActuator.doAction(3);}));
        steeringWheelCommands_Uturn.add(new Instruction(()->{steeringWheelActuator.doAction(-180);;}));
        steeringWheelCommands_Uturn.add(new Instruction(()->{steeringWheelActuator.doAction(0);;}));
        brakeCommands_Uturn.add(new Instruction(()->{brakeActuator.doAction(3);}));
        brakeCommands_Uturn.add(new Instruction(()->{brakeActuator.doAction(-3);}));

        Protocol acceleratorProtocol = new Protocol(accelatorCommands_Uturn);
        Protocol steeringWheelProtocol = new Protocol(steeringWheelCommands_Uturn);
        Protocol brakeProtocol = new Protocol(brakeCommands_Uturn);

        acceleratorActuator.setAcceleratorCommands(acceleratorProtocol);
        steeringWheelActuator.setSteeringWheelProtocol(steeringWheelProtocol);
        brakeActuator.setBrakeProtocol(brakeProtocol);

        adECU.autonomousDrive();
        assertEquals(3, acceleratorActuator.getAccelerationIntensity());
        assertEquals(0, steeringWheelActuator.getSteeringWheelInclination());
        assertEquals(-3, brakeActuator.getBrakeIntensity());
    }
}
```

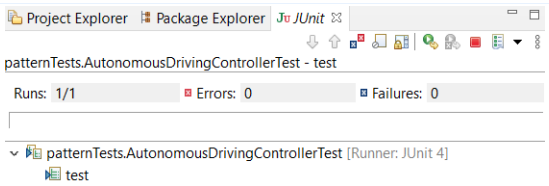

Figure 25: VehicleTest