Pattern-based Development

Walter F. Tichy

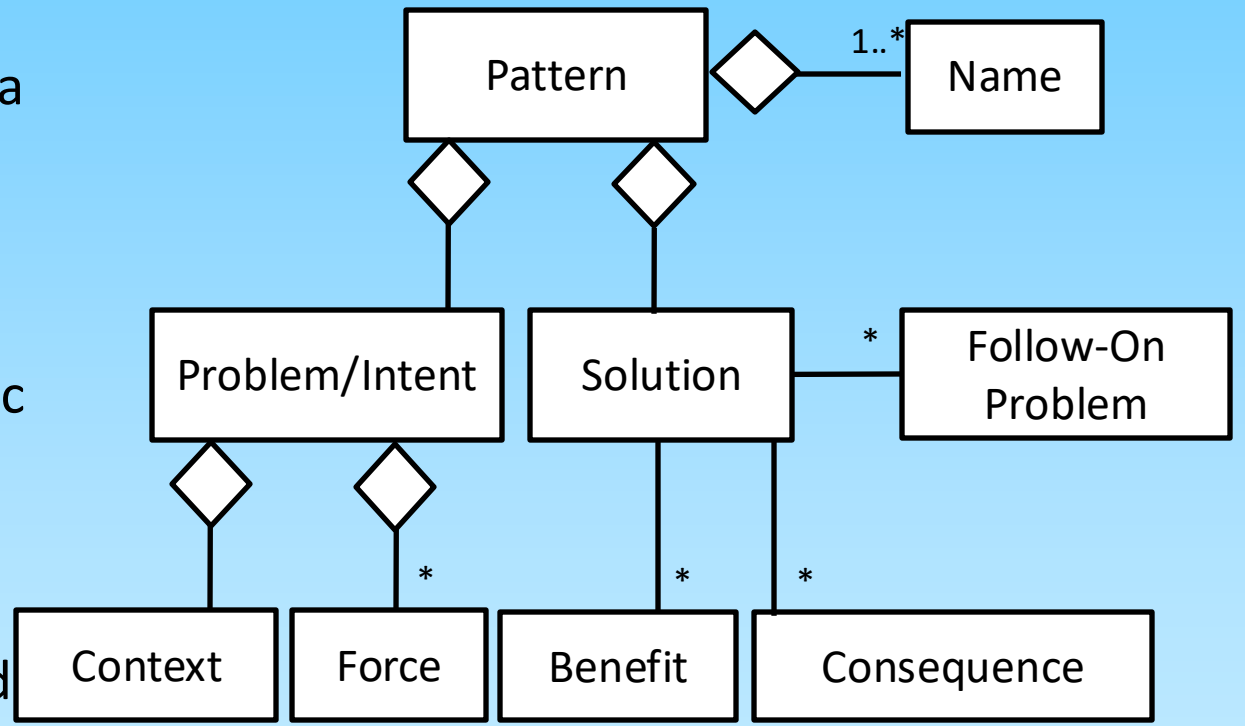
Karlsruhe Institute of Technology

Recap: What is a Design/Architectural Pattern?

A family of solutions for a design or architectural problem concerning the structure of software.

Modeling a Pattern in UML

- A pattern has one or more **names**, a **problem** and a **solution**
  - The **problem** class is elaborated in terms of a **context** and a set of **forces**
  - The **solution** resolves these forces with **benefits** and **consequences**
  - To be considered as a pattern, the solution must be applicable to more than one specific problem
- Solutions may generate **follow-on problems**
  - Follow-on problems can again be elaborated in terms of context and forces which may lead to the applicability of other patterns



Pattern-Based Development: Definition

- **Pattern-Based Development**
  - Model-based development that focuses on extensive use and reuse of patterns during analysis, design, and testing
  - Goal: Manage Complexity, reduce cost and time-to-market
  - Desirable: Applying patterns throughout the software lifecycle

- **Pattern Coverage**
  - Ideally, every element in the UML model and every element in the source code should be covered by a pattern
  - Desirable: 100% Coverage

The requirements provide clues for the use of patterns (1)

- *Text:* “must interface with an existing object”
  ⇒ **Adapter Pattern**
- *Text:* “must interface to several systems, some of them to be developed in the future”, “an early prototype must be demonstrated”, “must provide backward compatibility”
  ⇒ **Bridge Pattern**
- *Text:* “must interface to existing set of objects”, “must interface to existing API”, “must interface to existing service”
  ⇒ **Façade Pattern or Adapter Pattern**
- *Text:* “must be manufacturer independent”, “device independent”, “must support a family of products”
  ⇒ **Abstract Factory Pattern or Strategy Pattern**

The requirements provide clues for the use of patterns (2)

- *Text:* “complex structure”, “must have variable depth and width”
  - ⇒ **Composite Pattern, perhaps in combination with Visitor Pattern**
- *Text:* “must provide a policy independent from the mechanism”, “must allow to change algorithms at runtime”
  - ⇒ **Strategy Pattern**
- *Text:* “must be location transparent”
  - ⇒ **Proxy Pattern**
- *Text:* “must be updated”, “must be extensible”,
  - ⇒ **Observer Pattern** (MVC Architectural Pattern)

The requirements provide clues for the use of patterns (3)

- *Text:* “must accommodate many users”
  - ⇒ **Client-Server Pattern**
- *Text:* “provides higher-level functions”
  - ⇒ **Layers Pattern**
- *Text:* ”must be extensible by user”
  - ⇒ **Framework Pattern, Template Method Pattern, Strategy Pattern**
- *Text:* “must provide high throughput”,
  - ⇒ **Pipeline Pattern, master-worker pattern**

Exercise: For each pattern, think of potential textual clues in text.

Bumpers – Problem Statement [Bruegge]

Bumpers is a game where cars drive on a game board and can crash each other. In each collision, there is a winning car. The car that wins all collisions is the winner of the game. The player can start and stop the game. When the game is started, music is played.

A car can be either fast or slow. There is one car controlled by the player. The player can steer the direction of their car with the mouse and change its speed.

The game should be platform independent, and visualizes different parameters of the car, e.g. the speed, consumption, and location of the car. Different visualization types can be easily added.

When cars crash, there has to be a sound effect. The game should support different collisions and the determination of the collision winner should be changeable during gameplay.

Bumpers – Problem Statement

Bumpers is a game where cars drive on a game board and can crash each other. In each collision, there is a winning car. The car that wins all collisions is the winner of the game. The player can start and stop the game. When the game is started, music is played.

A car can be either fast or slow. There is one car controlled by the player. The player can steer the direction of their car with the mouse and change its speed.

The game should be platform independent, and visualizes different parameters of the car, e.g. the speed, consumption, and location of the car. **Different visualization types can be easily added.**

When cars crash, there has to be a sound effect. The game should support different collisions and the determination of the collision winner should be changeable during gameplay.

Bumpers – Problem Statement – Spot Patterns

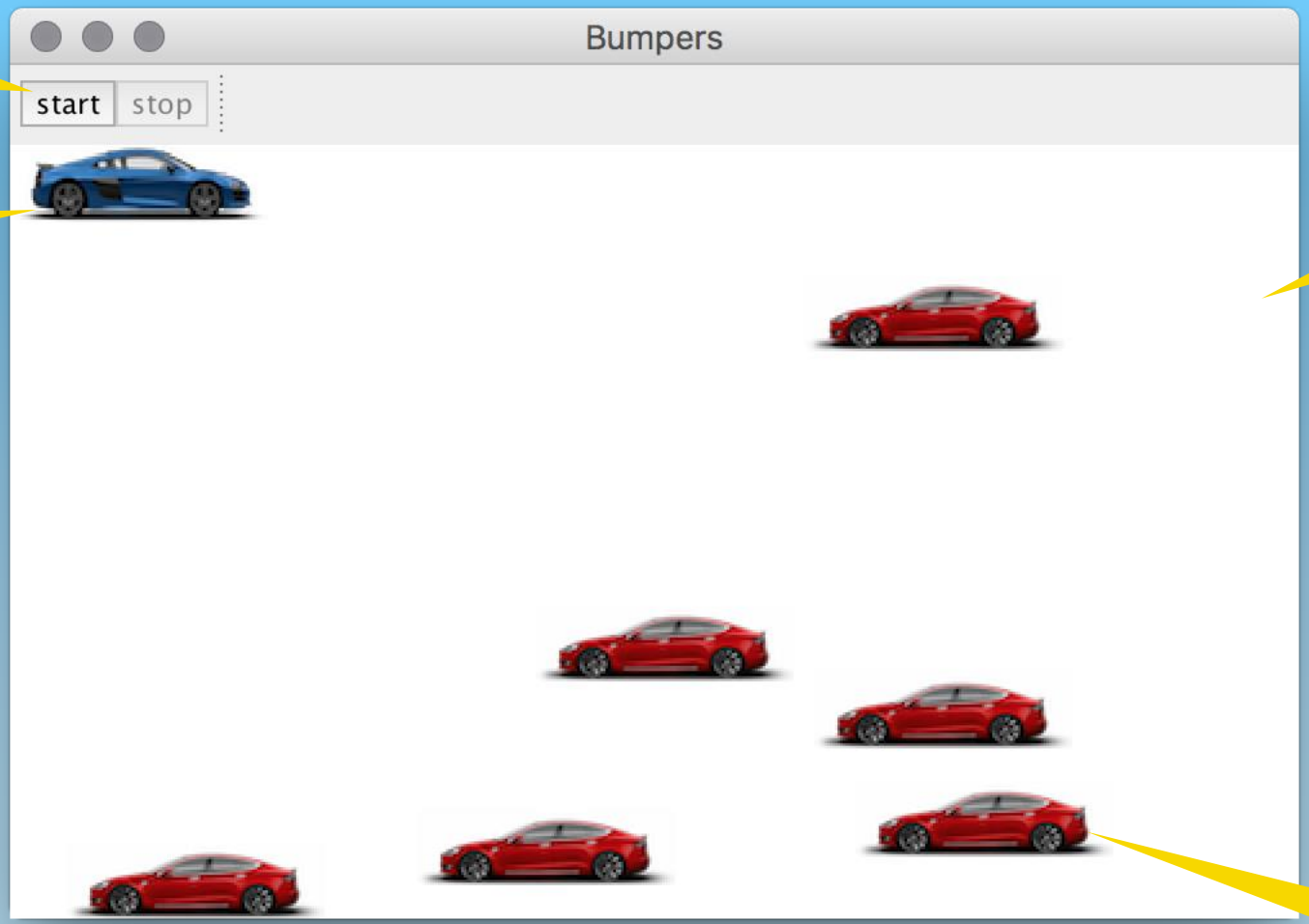
Bumpers is a game where cars drive on a game board and can crash each other. In each collision, there is a winning car. The car that wins all collisions is the winner of the game. The player can start and stop the game. When the game is started, music is played.

A car can be either fast or slow. There is one car controlled by the player. The player can steer the direction of their car with the mouse and change its speed.

The game should be **platform independent**, and visualizes different parameters of the car, e.g. the speed, consumption, and location of the car. **Different visualization types can be easily added.**

When cars crash, there has to be a sound effect. The game should **support different collisions** and the determination of the collision winner should **be changeable during gameplay**.

User Interface Design of Bumpers



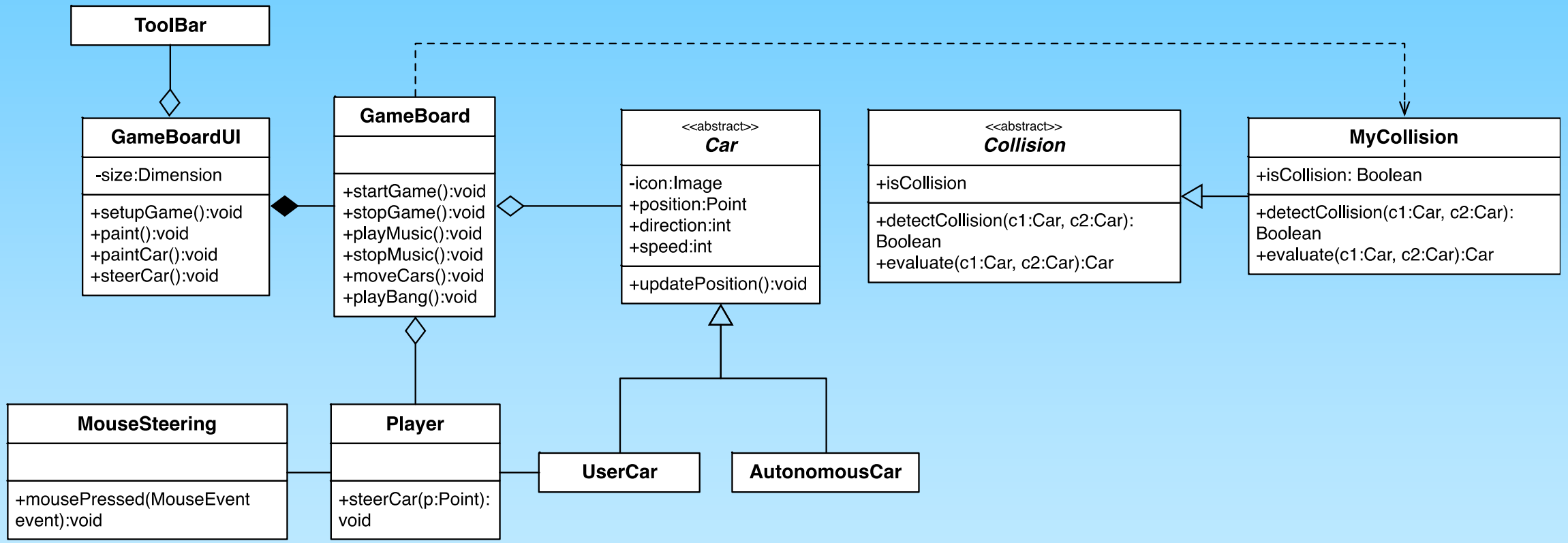
Toolbar

UserCar

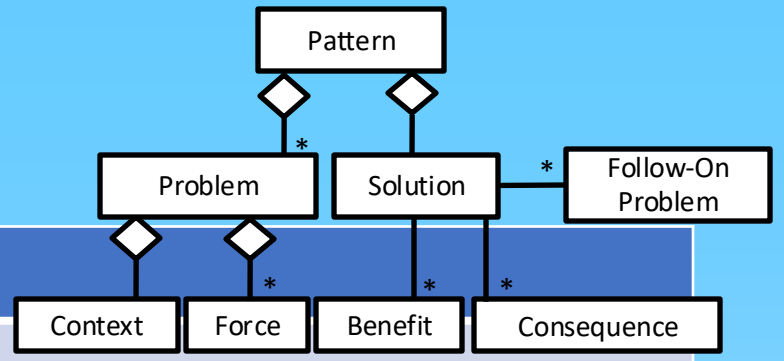
GameBoard

AutonomousCar

Initial Class Diagram

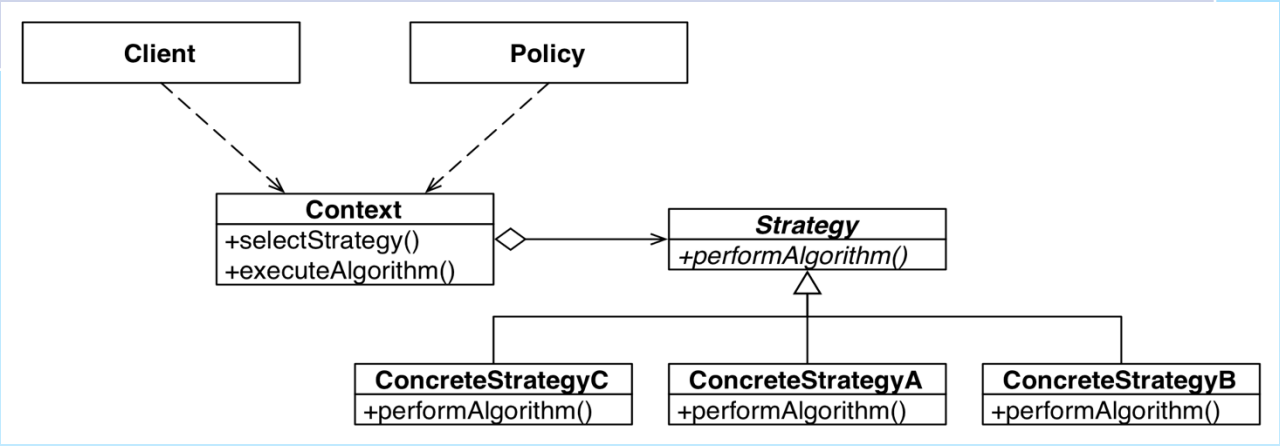


Applying the Strategy Pattern

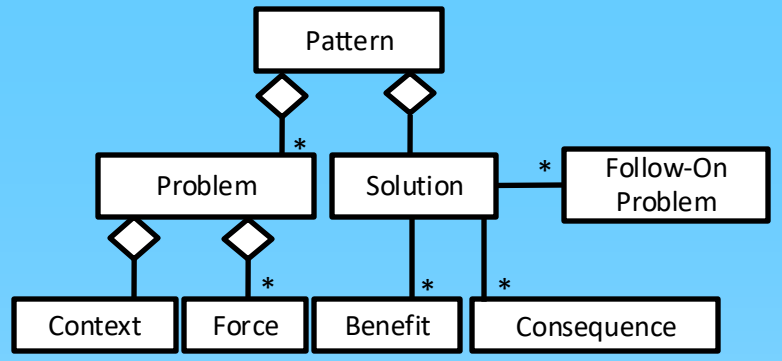


Problem	Switch between different algorithms at run-time
Context	Different algorithms exist for a specific task
Force	• Add a new algorithm without disturbing the application or other algorithms • A client needs to choose from multiple algorithms
Solution	• Extract different algorithms into separate classes: **ConcreteStrategy** • The **Client** accesses services provided by a **Context**: performAlgorithm() • The abstract ***Strategy*** class describes the interface that is common to all mechanisms that the **Context** can use • The **Policy** decides what **ConcreteStrategy** to use given the **Context**

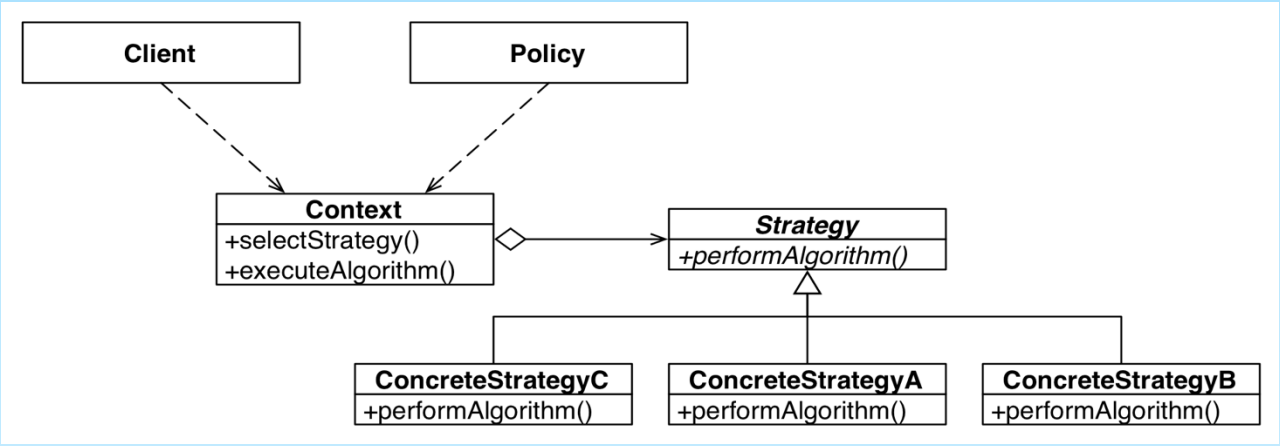
Continued on next slide



Strategy Pattern (ctd)

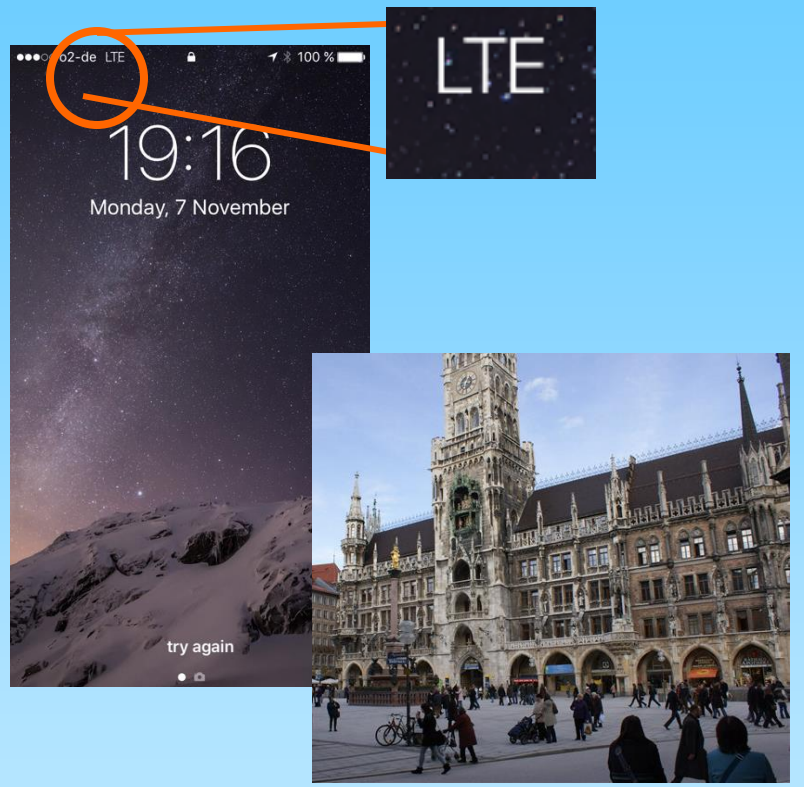


Benefit	• New algorithms can be added without modifying **Context** or **Client** • **ConcreteStrategies** can be substituted transparently from *Context*
Consequence	• **Policy** decides which ***Strategy*** is best, given the current circumstances (e.g., type of collision) • The **Client** is not aware of the **ConcreteStrategies**
Follow-On Problem	Code becomes more complex the more **ConcreteStrategies** are added



Strategy Pattern Real Life Application Example

If WiFi available, use WiFi …



… otherwise, use mobile data

Bumpers – Problem Statement

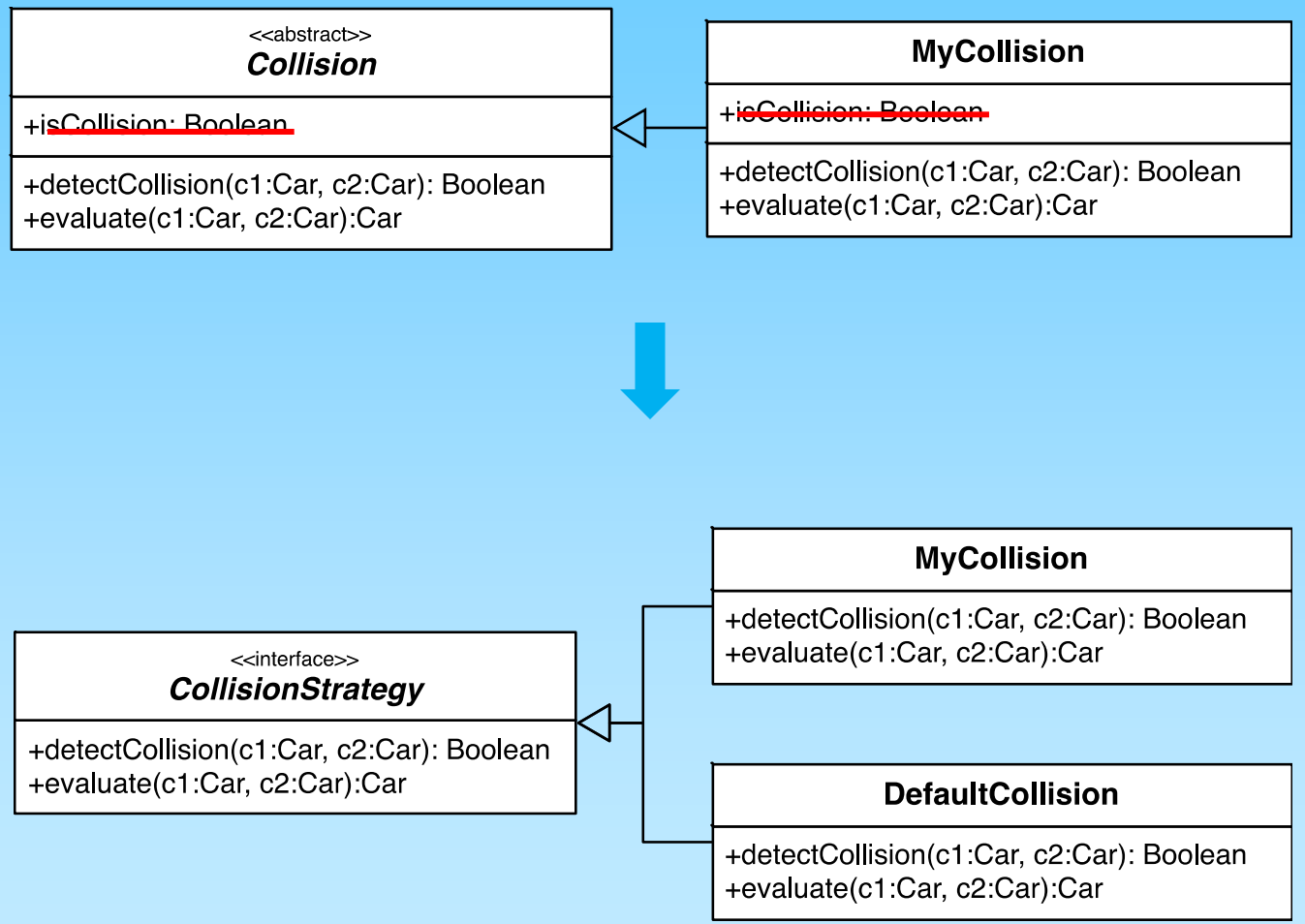
Bumpers is a game where cars drive on a game board and can crash each other. In each collision, there is a winning car. The car that wins all collisions is the winner of the game. The player can start and stop the game. When the game is started, music is played.

A car can be either fast or slow. There is one car controlled by the player. The player can steer the direction of their car with the mouse and change its speed.

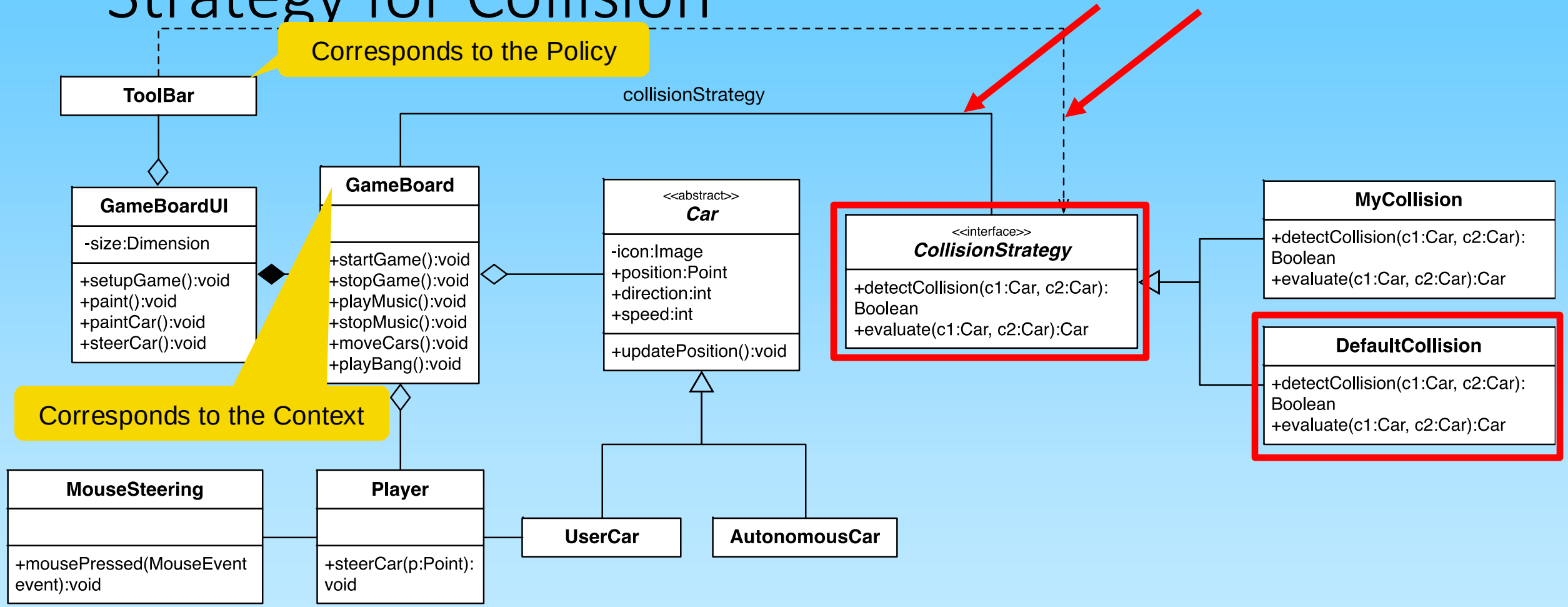
The game should be platform independent, and visualizes different parameters of the car, e.g. the speed, consumption, and location of the car.

When cars crash, there has to be a sound effect. **The game should support different collisions and the determination of the collision winner should be changeable during gameplay.**

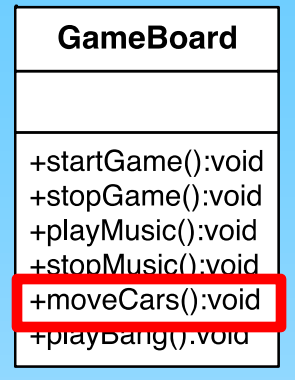
Hint: Use the Strategy Pattern



Strategy for Collision



Setting the DefaultCollision Strategy - Code

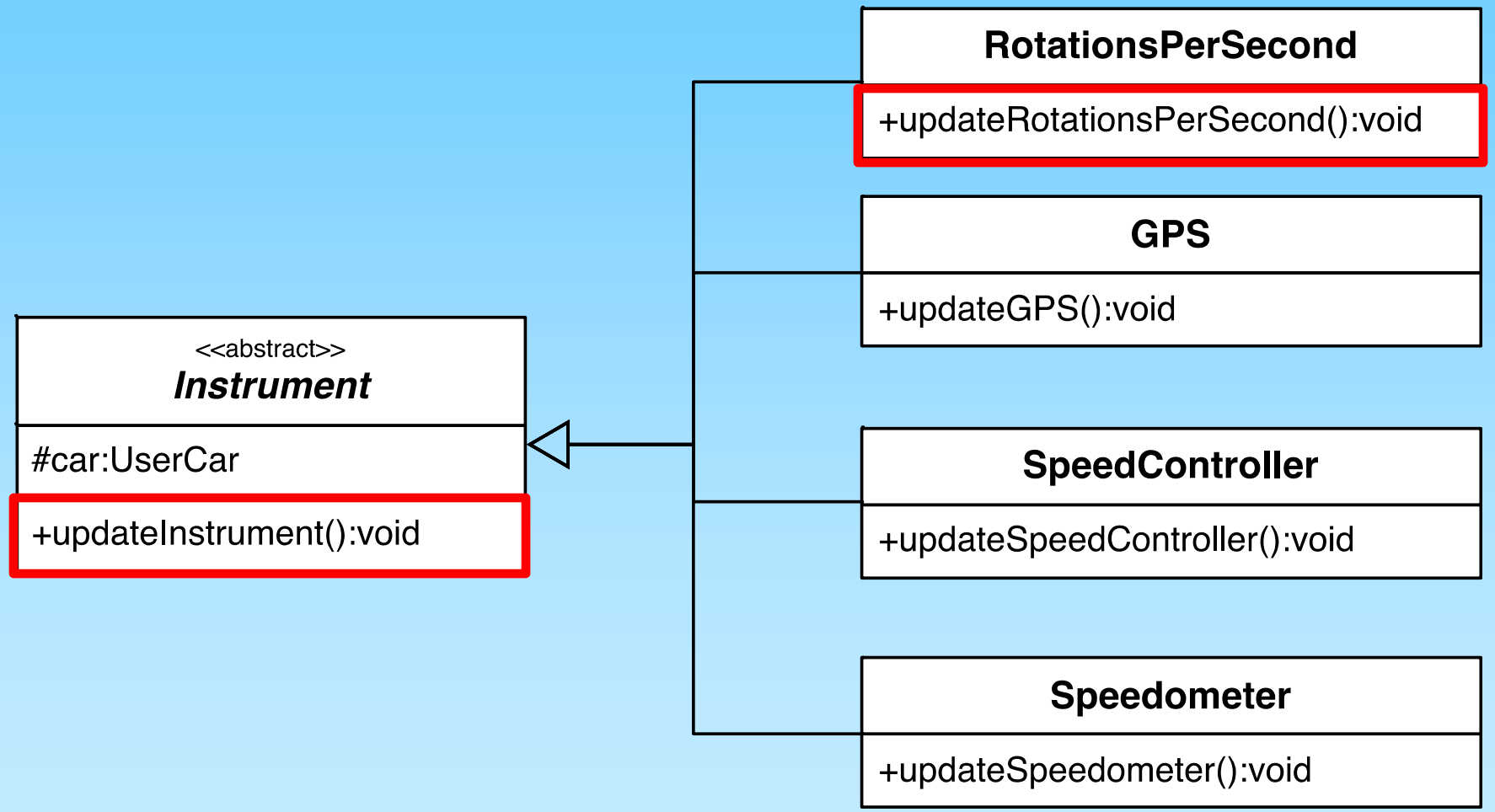



```
public class GameBoard {
...
private CollisionStrategy collisionStrategy = new DefaultCollision();
...
        public void moveCars() {
                ...
                for (Car car : cars) {
                        ...
                        if(collisionStrategy.detectCollision(getPlayerCar(), car)) {
                                collisionStrategy.evaluate(getPlayerCar(), car);
                                audioPlayer.playBangAudio();
                        }
                }
        }

        public CollisionStrategy getCollisionStrategy() {
                return collisionStrategy;
        }

        public void setCollisionStrategy(CollisionStrategy collisionStrategy) {
                this.collisionStrategy = collisionStrategy;
        }
}
```

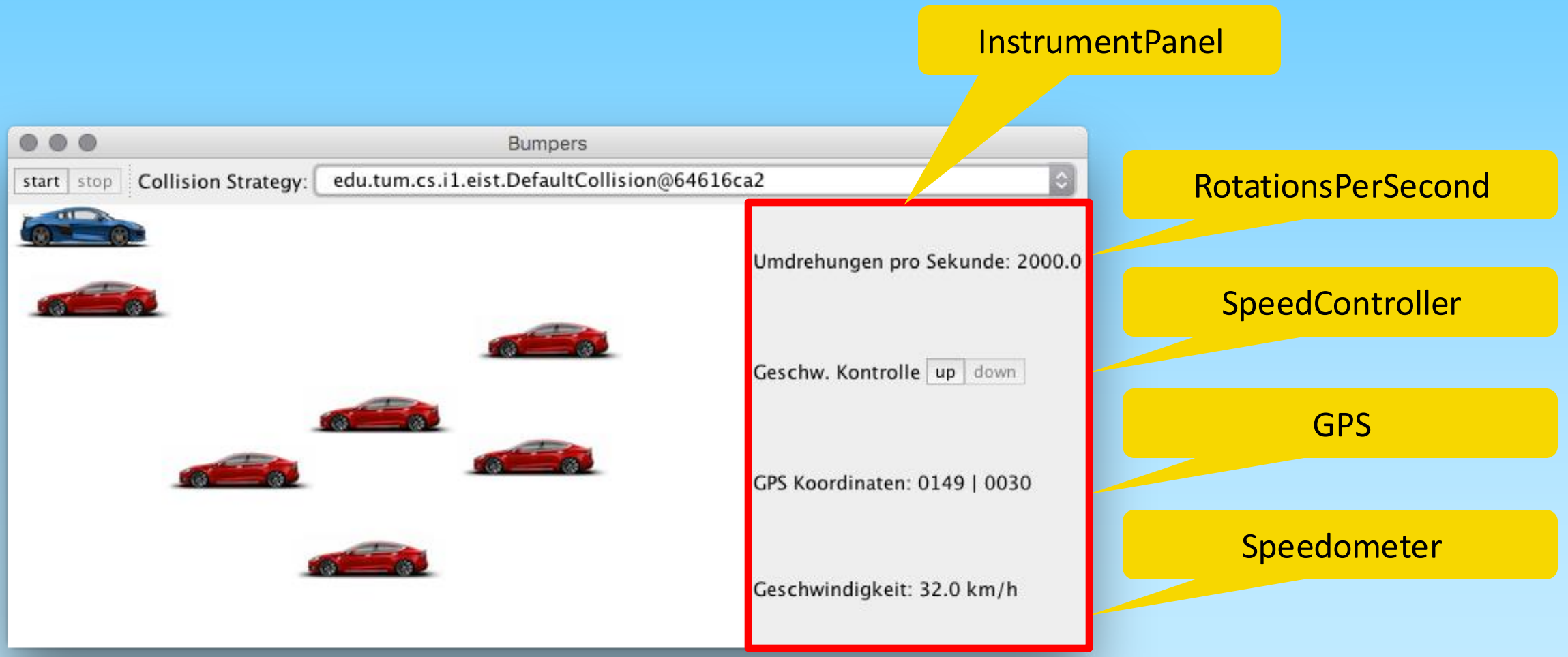
Next steps:

- The player can change the car’s speed
- The speed, consumption, and location of the player’s car is visualized in an instrument panel

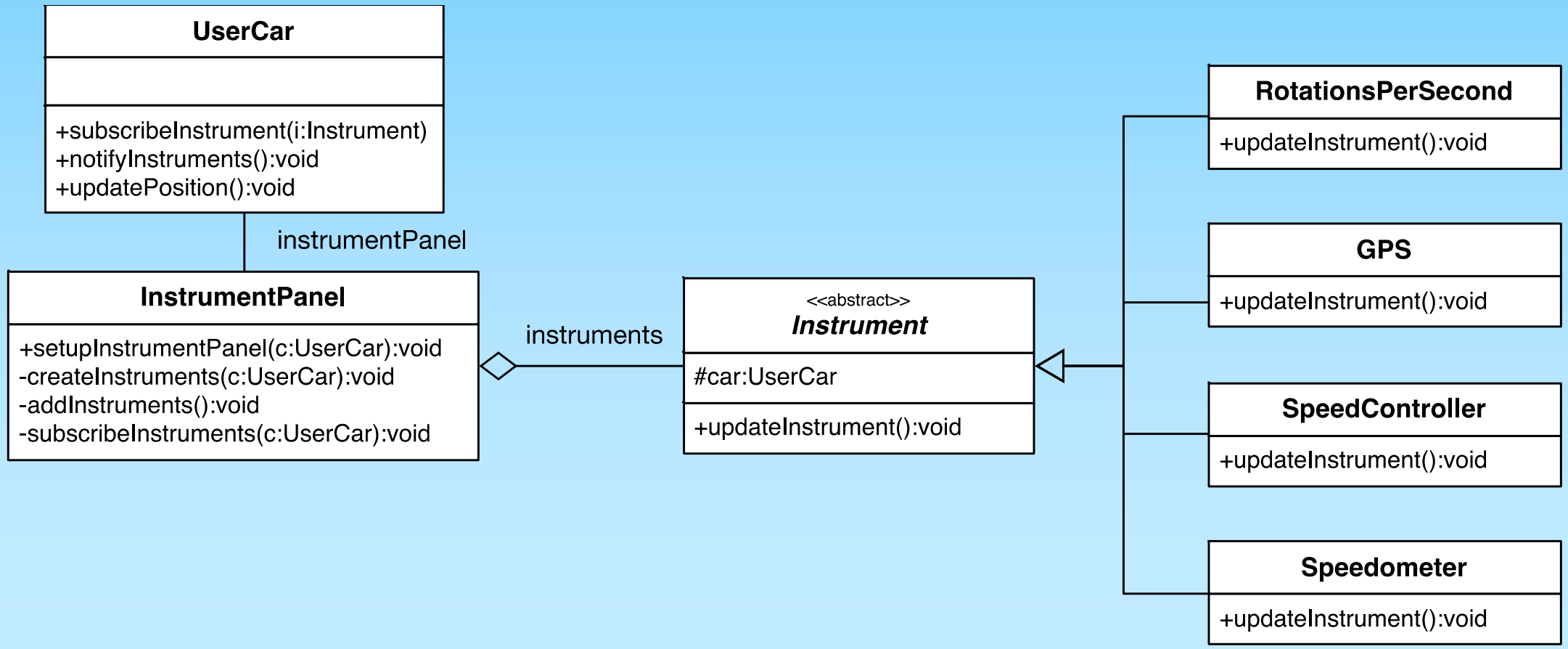
Instruments available, but incompatible



Use Observer for Instrument panel

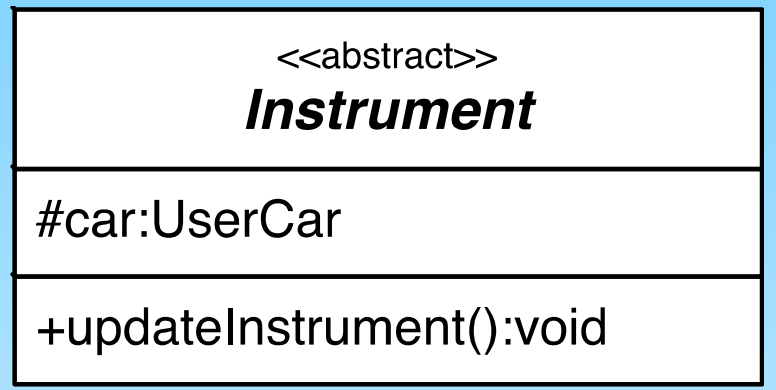


Hint: use observer, extend signatures of instruments

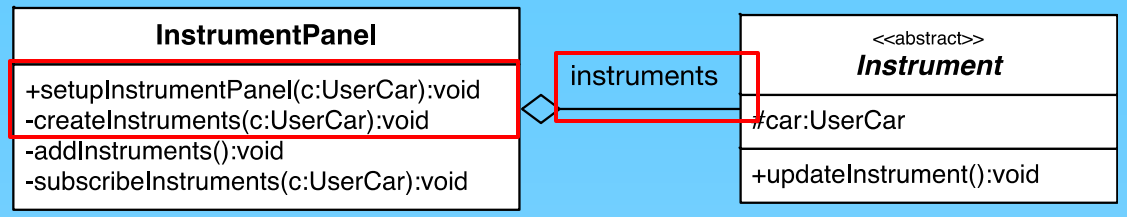


Hint: Introduction of the abstract class Instrument

```
public abstract class Instrument extends JPanel {

        protected UserCar car;


        public Instrument(UserCar car) {
                this.car = car;
        }

        public abstract void updateInstrument();


}
```

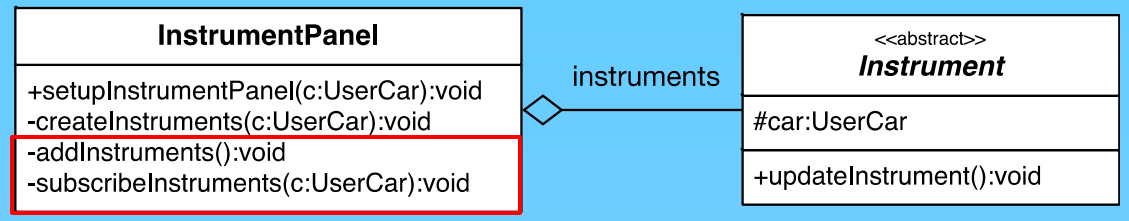


InstrumentPanel Pt. 1




```
public class InstrumentPanel extends JToolBar {

    public List<Instrument> instruments = new ArrayList<Instrument>();

    public InstrumentPanel() {
            super(JToolBar.VERTICAL);
            setFloatable(false);
    }

    public void setupInstrumentPanel(UserCar userCar) {
            createInstruments(userCar);
            subscribeInstruments(userCar);
            addInstruments();
    }

    private void createInstruments(UserCar car) {
            if (instruments.size() == 0) {
                    instruments.add(new RotationsPerSecond(car));
                    instruments.add(new SpeedController(car));
                    instruments.add(new GPS(car));
                    instruments.add(new Speedometer(car));
            }
    }
```

InstrumentPanel Pt. 2



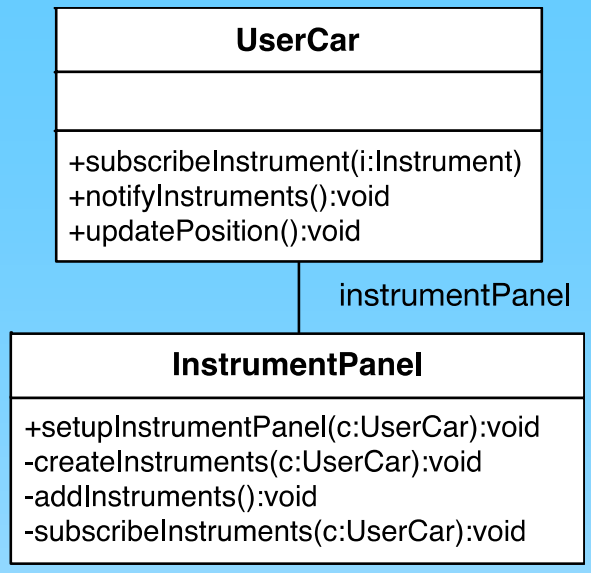

```
public class InstrumentPanel extends JToolBar {

        public List<Instrument> instruments = new ArrayList<Instrument>();


        ...

        private void addInstruments() {
                for (Instrument instrument: instruments) {
                        add(instrument);
                }
        }

        private void subscribeInstruments(UserCar car) {
                for (Instrument instrument: instruments) {
                        car.subscribeInstrument(instrument);
                }
        }
}
```

Adding InstrumentPanel to the User Interface

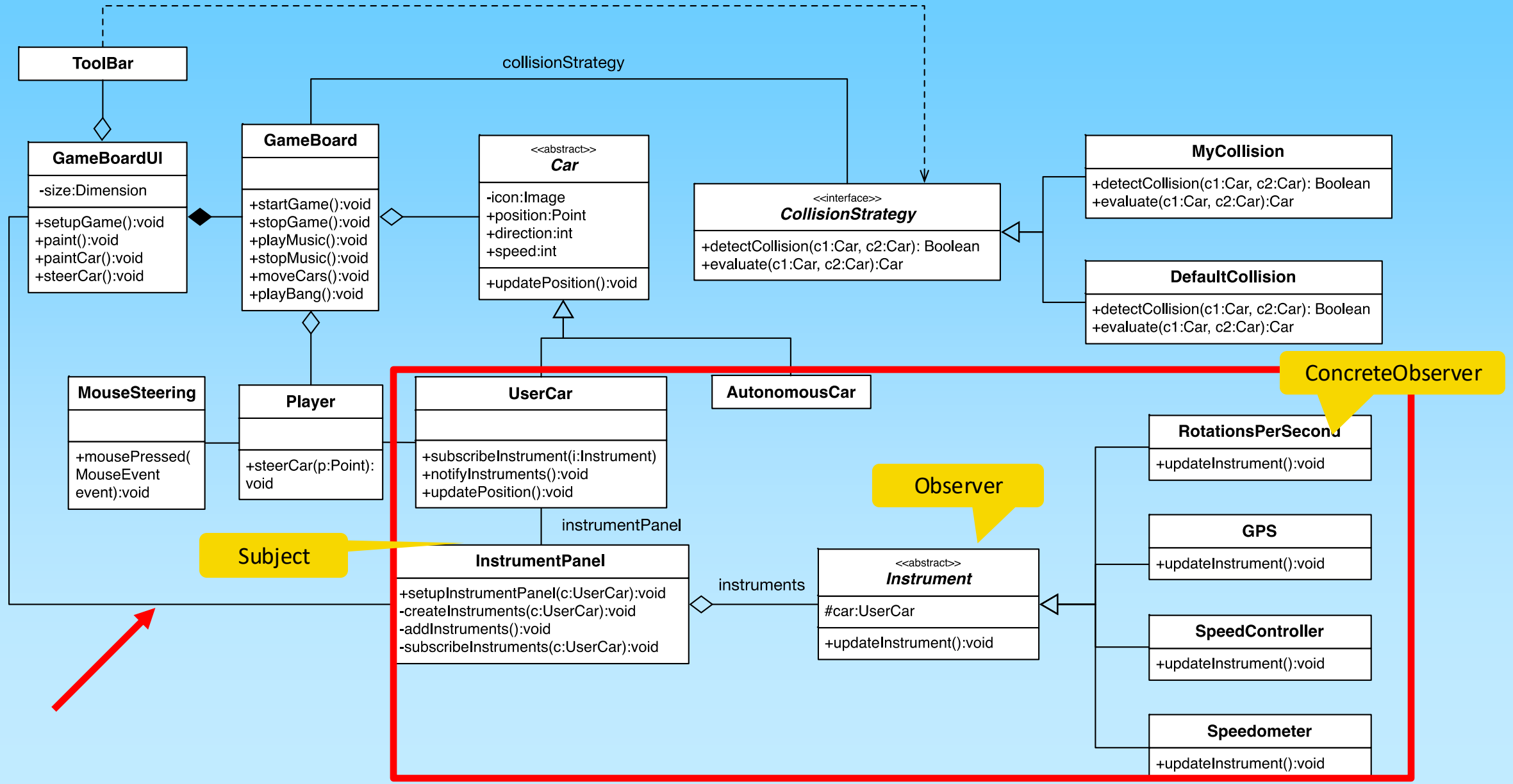
```
public class BumpersWindow extends JFrame {

        …

        public BumpersWindow() {
                super("Bumpers");

                …

                getContentPane().setLayout(new BorderLayout());
                content.add(toolBar, BorderLayout.NORTH);
                content.add(gameBoardUI, BorderLayout.CENTER);
                content.add(gameBoardUI.gameBoard.getPlayerCar().getInstrumentPanel(), BorderLayout.EAST);
                getContentPane().add(content, BorderLayout.CENTER);
                toolBar.setupStrategyBox();
        }

        …
}
```

Updating the Instruments

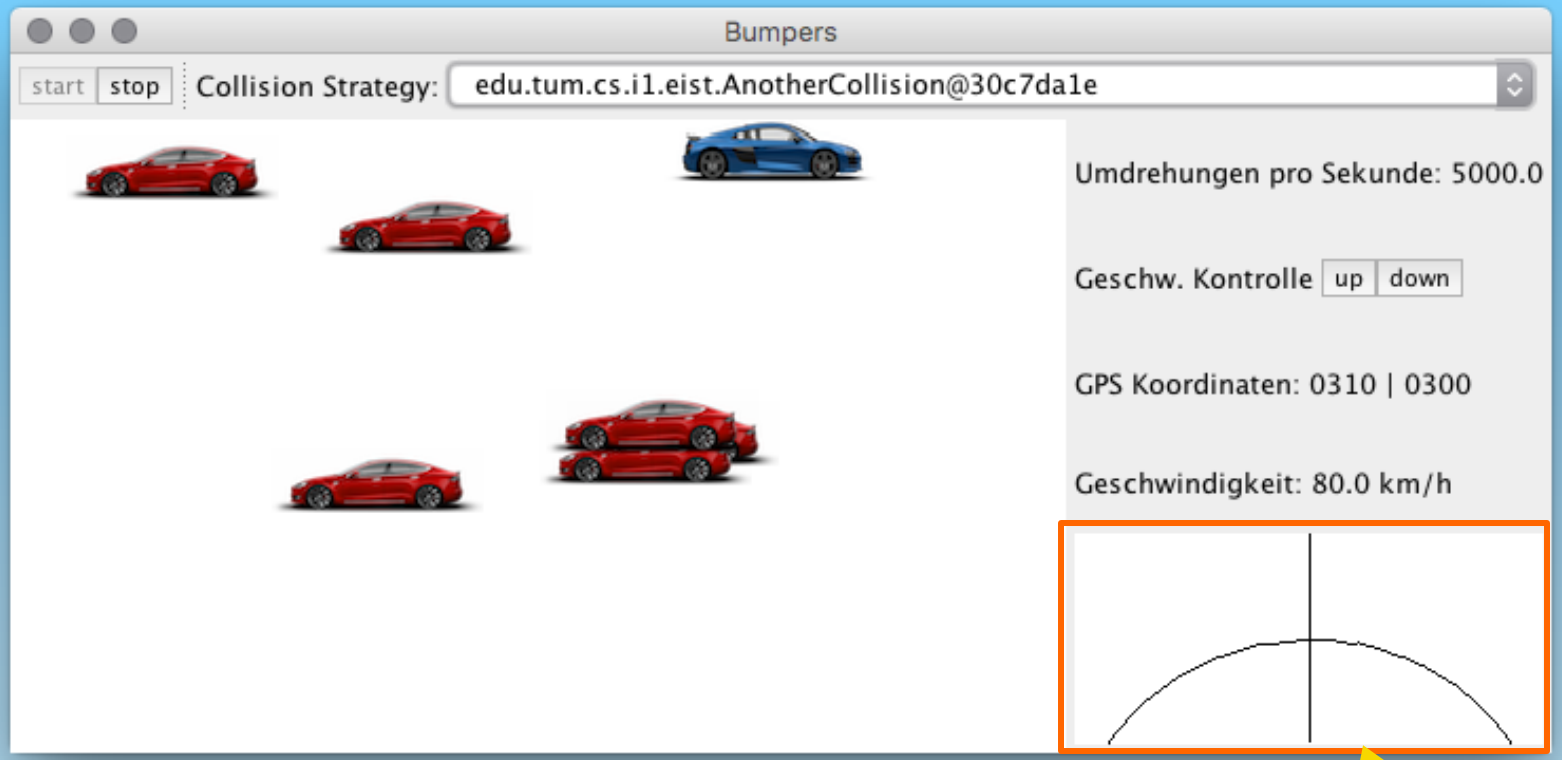
```
public class UserCar extends Car {
        ...
        private InstrumentPanel instrumentPanel;


        public UserCar(int max_x, int max_y) {
                super(max_x, max_y);
                instrumentPanel = new InstrumentPanel();
                instrumentPanel.setupInstrumentPanel(this);
                setBody(DEFAULT_USER_CAR_IMAGE);
        }

        public void subscribeInstrument(Instrument instrument) {
                if (instrument != null && !this.instrumentPanel.instruments.contains(instrument)) {
                this.instrumentPanel.instruments.add(instrument);
                }
        }

        public void notifyInstruments() {
                for (Instrument instrument: this.instrumentPanel.instruments) {
                        instrument.updateInstrument();
                }
        }


        @Override
        public void updatePosition(int max_x, int max_y) {
                super.updatePosition(max_x, max_y);
                notifyInstruments();
        }

}
```



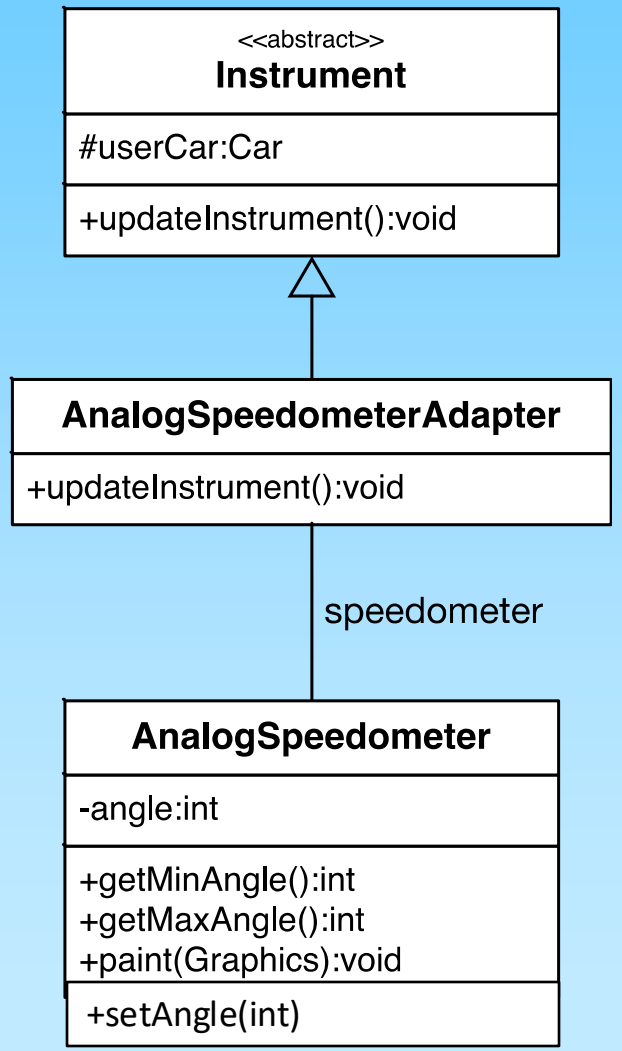
Class model with instrument panel



Last task: add an analog speedometer

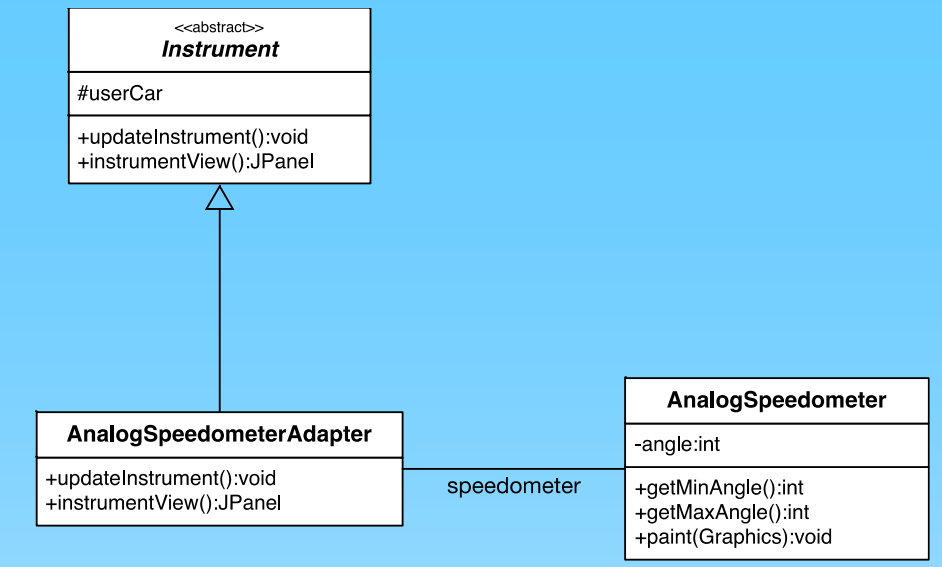


Hint: use an adapter for the analog speedometer

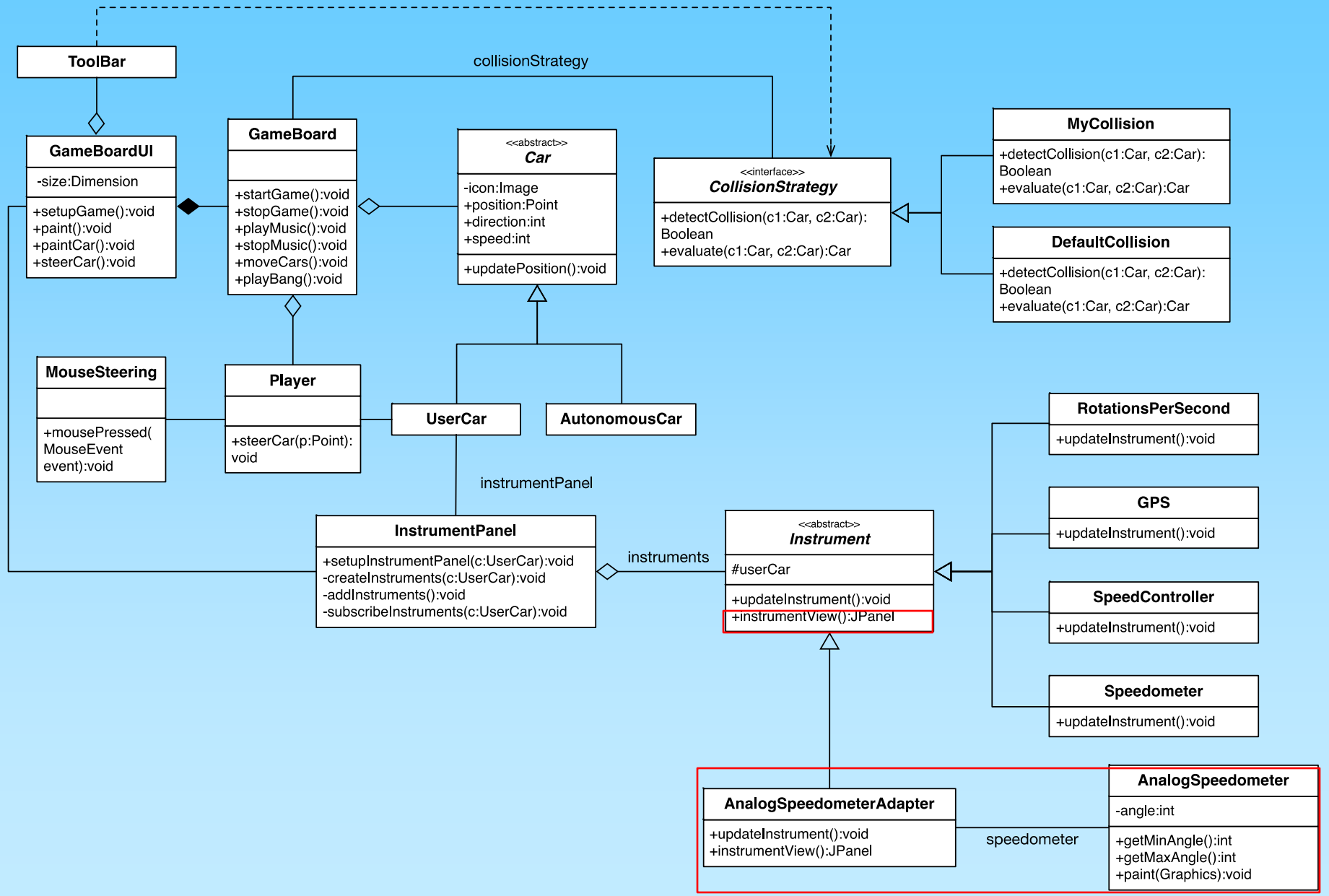


AnalogSpeedometerAdapter

```
public class AnalogSpeedometerAdapter extends Instrument {
        private AnalogSpeedometer speedometer = new AnalogSpeedometer();
        private int speed;

        public AnalogSpeedometerAdapter(UserCar userCar){
                super(userCar);
                updateInstrument();
        }

        @Override
        public void updateInstrument() {
                if(this.speed != car.getSpeed()){
                        this.speed = car.getSpeed();
                        double percent = (double)car.MAX_SPEED/100*this.speed;
                        int angle = (int)(speedometer.getMaxAngle()*percent);
                        this.speedometer.setAngle(angle);
                }
        }


        @Override
        public JPanel instrumentView() {
                return speedometer;
        }
}
```



Class diagram with analog speedometer



Summary

- Pattern-Based Development: Applying Patterns throughout the Software Lifecycle. Model-Based Development that focuses on extensive use and reuse of patterns during analysis, design and testing.
  - Application of the **Strategy Pattern**
  - Application of the **Observer Pattern**
  - Application of the **Adapter Pattern**
- Pattern Coverage: Ideally, every Model element and every element in the code is covered by a pattern. Desired: 100% Coverage