Strategy Pattern

Michal Moravik, SD20w2

Name: Strategy Pattern

Category: Behavioral

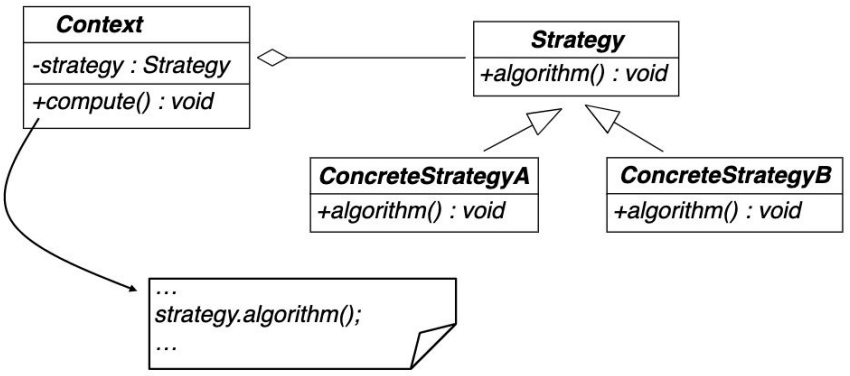
Intent: To create classes representing various behaviours (also called “strategies”) and to be able to easily switch between these strategies.

Motivation: We want to use this pattern if we have a lot of similar classes that differ in the way they execute their behaviour.

Real case scenario:
We want to be able to change strategies during the runtime to create functionality as, for example, Google Maps has. There are several ways of travel from point A to point B, it can be a car, bus, … These ways of travel can be changed easily without affecting the main functionality (going from A to B) although each strategy does a different action (chooses a different route).

The main functionality will therefore be dependent on the abstraction of those strategies and execute the function declared by the abstraction with a different implementation for each of the strategies.

UML:



Implementation:

```java
public interface IVehicle {
    void transport();
}
```

```java
public class HotelShuttleBus implements IVehicle {

    @Override
    public void transport() {
        System.out.println("Transporting using hotel shuttle bus");
    }
}
```

```java
public class PersonalVehicle implements IVehicle {
    @Override
    public void transport() {
        System.out.println("Transporting using personal car");

    }
}
```

```java
public class Taxicab implements IVehicle {
    @Override
    public void transport() {
        System.out.println("Transporting using taxicab");
    }
}
```

```java
public class Traveling {
    private IVehicle vehicle;

    public Traveling(IVehicle vehicle) {
        this.vehicle = vehicle;
    }

    public void changeVehicle(IVehicle vehicle) {
        this.vehicle = vehicle;
    }

    public void travel() {
        vehicle.transport();
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        IVehicle taxicab = new Taxicab();
        IVehicle personalVehicle = new PersonalVehicle();
        IVehicle hotelShuttleBus = new HotelShuttleBus();

        // default one - creating a new Traveling object
        Traveling traveling = new Traveling(personalVehicle);
        traveling.travel();

        // change of the vehicle, but still calling travel() method
        traveling.changeVehicle(taxicab);
        traveling.travel();
        traveling.changeVehicle(hotelShuttleBus);
        traveling.travel();
    }
}
```

```
Transporting using personal car
Transporting using taxicab
Transporting using hotel shuttle bus
```

Consequences:

Pros:

- Swapping strategies during the runtime
- Isolation of the strategy's behaviour from the code that uses it
- New strategies can be introduced without changing the context
- Eliminates large conditional statements (like switch case)

Cons:

- The number of objects is increased
- To select a proper strategy, you must know each of them

Known uses:

- Could be used in GPS tracking devices
- Could be used in games (switching spells for example)

Related patterns:

- Command Pattern and Strategy Pattern are often mistaken. They have different intents though. Command converts operations into objects and to store the operation's parameters as fields of the object. Thanks to this you can queue it, undo it, and more.

  Strategy just describes a different way of doing the same thing, letting you swap the concrete implementation of the same functionality.