Strategy DP

Deliverable 2

Macavei Rares - 30432

Brief definition

- *behavioural DP*
- *The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it*

When to use

When you want to be able to change the behavior at run-time dynamically.

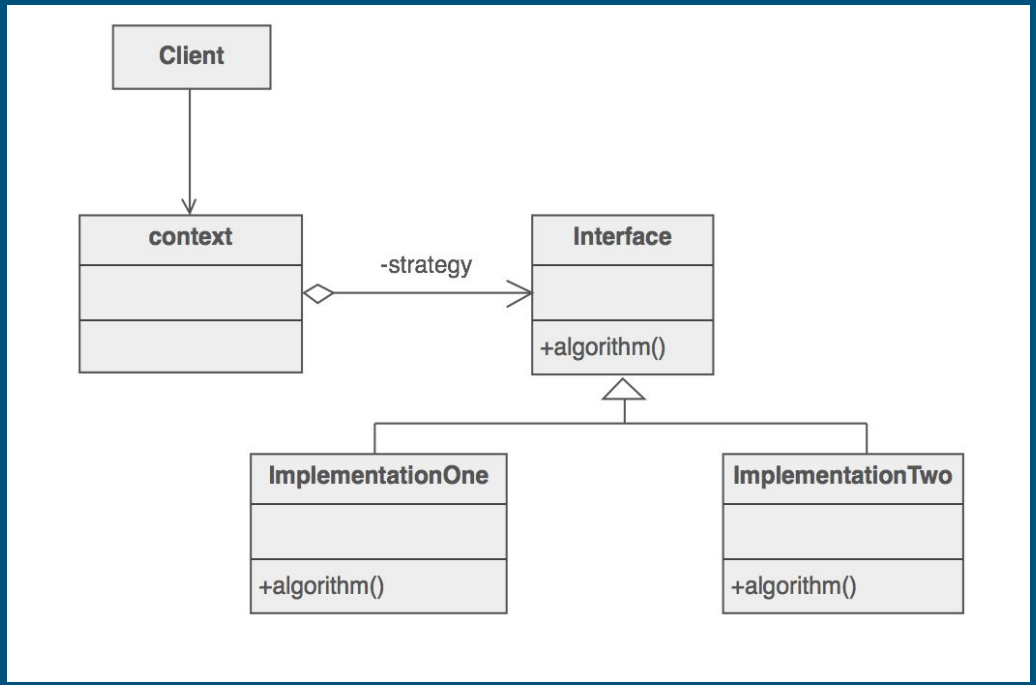
Pros:

1. Prevents the conditional statements. (switch, if, else…)
2. The algorithms are loosely coupled with the context entity. They can be changed/replaced without changing the context entity.
3. Easy to extend

Cons:

1. Clients must know about the existence of different strategies and a client must understand how the Strategies differ
2. It increases the number of objects in the application.

Class Diagram



Example

```java
import java.util.ArrayList;

@FunctionalInterface
interface BillingStrategy {
    // use a price in cents to avoid floating point round-off error
    int getActPrice(int rawPrice);

    //Normal billing strategy (unchanged price)
    static BillingStrategy normalStrategy() {
        return rawPrice -> rawPrice;
    }

    //Strategy for Happy hour (50% discount)
    static BillingStrategy happyHourStrategy() {
        return rawPrice -> rawPrice / 2;
    }
}

class Customer {
    private final ArrayList<Integer> drinks = new ArrayList<>();
    private BillingStrategy strategy;

    public Customer(BillingStrategy strategy) {
        this.strategy = strategy;
    }

    public void add(int price, int quantity) {
        this.drinks.add(this.strategy.getActPrice(price*quantity));
    }

    // Payment of bill
    public void printBill() {
        int sum = this.drinks.stream().mapToInt(v -> v).sum();
        System.out.println("Total due: " + sum / 100.0);
        this.drinks.clear();
    }

    // Set Strategy
    public void setStrategy(BillingStrategy strategy) {
        this.strategy = strategy;
    }
}
```

```java
public class StrategyPattern {
    public static void main(String[] arguments) {
        // Prepare strategies
        BillingStrategy normalStrategy    = BillingStrategy.normalStrategy();
        BillingStrategy happyHourStrategy = BillingStrategy.happyHourStrategy();

        Customer firstCustomer = new Customer(normalStrategy);

        // Normal billing
        firstCustomer.add(100, 1);

        // Start Happy Hour
        firstCustomer.setStrategy(happyHourStrategy);
        firstCustomer.add(100, 2);

        // New Customer
        Customer secondCustomer = new Customer(happyHourStrategy);
        secondCustomer.add(80, 1);
        // The Customer pays
        firstCustomer.printBill();

        // End Happy Hour
        secondCustomer.setStrategy(normalStrategy);
        secondCustomer.add(130, 2);
        secondCustomer.add(250, 1);
        secondCustomer.printBill();
    }
}
```

Check List ✓

1. Identify an algorithm (i.e. a behavior) that the client would prefer to access through a "flex point".
2. Specify the signature for that algorithm in an interface.
3. Write the alternative implementation details in derived classes.
4. Clients of the algorithm couple themselves to the interface.