Observer Pattern

Michal Moravik, SD20w2

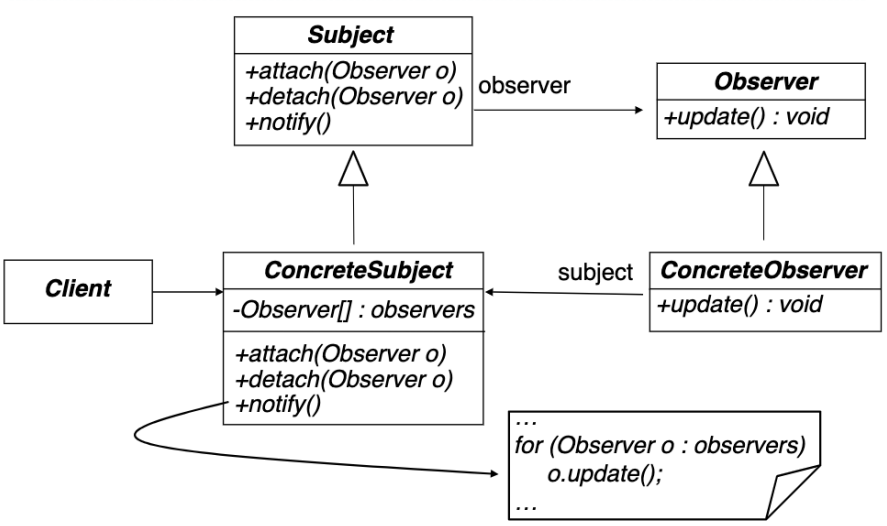
Name: Observer Pattern

Category: Behavioral

Intent: Let's us define observable and observers (one-to-many relationship). If observable changes its state, observers are automatically notified.

Motivation: We use the pattern if we want to observe a specific subject (observable) and notify (send a message, perform an action, …) observers when the state of the subject changes. Sometimes, the subject is called the publisher and observers subscribers.

UML:



Implementation:
Below you can find a code which simulates a situation in the auction house. The auction house holds a list of all observers in order to be able to notify them when something changes. One of the observers bids whenever the current price of an item is less than 80, the other one when the price is less than 60 (since he has less money). So when the price is set to 70, you can see that one of the observers leaves the bid while the other one bids more.

```java
public interface ISubject {
    void attach(IObserver os);
    void detach(IObserver os);
    void notifyObservers();
}
```

```java
public class ConcreteSubject implements ISubject {
    List<IObserver> listOfObservers;
    private int price;

    public ConcreteSubject(int price) {
        this.listOfObservers = new ArrayList<>();
        this.price = price;
    }

    @Override
    public void attach(IObserver os) { this.listOfObservers.add(os); }

    @Override
    public void detach(IObserver os) { this.listOfObservers.remove(os); }

    @Override
    public void notifyObservers() {
        for (IObserver observer : this.listOfObservers) {
            observer.update(this.price, this);
        }
    }

    public int getPrice() { return price; }

    public void setPrice(int price) { this.price = price; }
}
```

```java
public interface IObserver {
    void update(int price, ISubject observable);
}
```

```java
public class ConcreteObserver2 implements IObserver {
    public ConcreteObserver2() {
    }

    @Override
    public void update(int price, ISubject subject) {
        if (price < 80) {
            ((ConcreteSubject) subject).setPrice(price + 20);
            System.out.println("I am bidding more! " + ((ConcreteSubject) subject).getPrice());
        } else {
            System.out.println("Leaving the bid!");
        }
    }
}
```

```java
public class ConcreteObserver implements IObserver {
    public ConcreteObserver() {}

    @Override
    public void update(int price, ISubject subject) {
        if (price < 60) {
            System.out.println("I am bidding more!");
            ((ConcreteSubject) subject).setPrice(price + 20);
        } else {
            System.out.println("Leaving the bid!");
        }
    }
}
```

```java
public class Main {

    public static void main(String[] args) throws IOException {
        IObserver observer1 =  new ConcreteObserver();
        IObserver observer2 = new ConcreteObserver2();

        ConcreteSubject subject = new ConcreteSubject(70);
        subject.attach(observer1);
        subject.attach(observer2);

        subject.notifyObservers();
    }
}
```

```
Leaving the bid!
I am bidding more! 90
```

Consequences:

Pros:

- All subscribers are notified about the changed state. Change at one place triggers some automated process which is handy functionality.
- We can establish this relation at runtime

- Introducing a new subscriber does not require changes in the publisher's code and vice versa

Cons:

- It is a monolithic approach, and so, future scaling is a problem
- Subscribers are often notified in random order

Known uses:

- GUI listeners
- MVC frameworks. When the action of the UI field is triggered, the controller will be notified.