State Pattern

Michal Moravik, SD20w2

Name: State Pattern

Category: Behavioral

Intent: To have an object which changes its behaviour when its internal state changes.

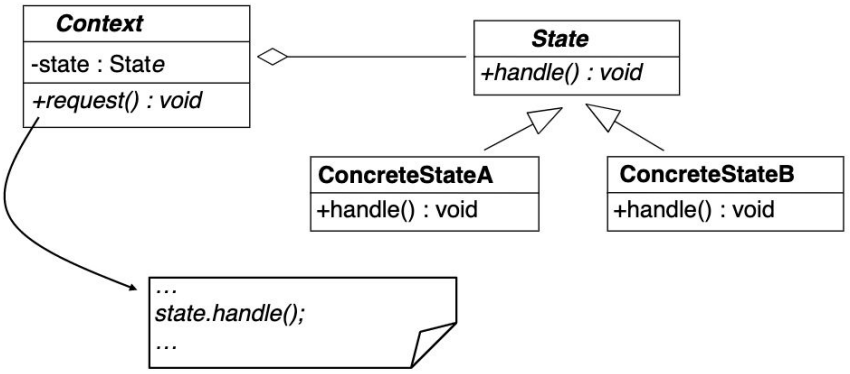
Motivation: In order to have an object which has states and these states influence the object's behaviour, we would need to use conditional statements (like *if* or *switch*). In the switch statement, the cases would represent states. Based on a different state, the player does a different action after pressing the "Play" button.

```
public class Player {
    String state;

    public void pressPlayButton() {
        switch (state) {
            case "playing":
                // show message that the player already plays
                // so clicking "play" won't do anything
                playerAlreadyActiveMessage();
                break;
            case "paused":
                playMusic();
                break;
            case "deactivated":
                // show message that there is an internet connection error
                showInternetConnectionErrorMessage();
                break;
        }
    }
```

In order to avoid multiple long conditional statements, we implement the State Pattern.

UML:



Implementation:

```java
public abstract class State {
    Player player;

    State(Player player) {
        this.player = player;
    }

    public abstract void onPlay();
}
```

```java
public class PauseState extends State {

    PauseState(Player player) {
        super(player);
    }

    @Override
    public void onPlay() {
        System.out.println("Play pressed!");
        player.changeState(new PlayState(player));
    }
}
```

```java
public class PlayState extends State {

    PlayState(Player player) {
        super(player);
    }

    @Override
    public void onPlay() {
        System.out.println("You already play music!");
    }
}
```

```java
public class Player {
    private State state;

    public Player() {
        this.state = new PauseState(this);
    }

    public void changeState(State state) {
        this.state = state;
    }

    public State getState() {
        return state;
    }
}
```

```java
public class Client {
    public static void main(String[] args) {
        Player player = new Player();
        player.getState().onPlay();

        player.changeState(new PlayState(player));
        player.getState().onPlay();
    }
}
```

```
Play pressed!
You already play music!
```

The player has a “pause” state by default so after we call “play” method, it starts playing (see the output). When we change the state to “play” state and then call “play” method, the system prints out the message instead (see the output). This is because the player was already playing and so, calling play method should do nothing but show the message.

Consequences:

Pros:

- Can introduce new states without changing existing state classes or the context
- Remove massive conditionals
- Separation of states into different classes - single responsibility principle

Cons:

- Can be easily an overkill if the object’s state rarely changes or if the conditional statements are too small. It is worthy when the states are highly used and there either massive condition statements or too many of them.
- A large number of objects is necessary (for each state) - taking memory

Known uses:

- Use when having an object that behaves differently based on its current state

- Use when your class includes massive conditionals that alter how the class behaves based on the current values of some field

Related patterns:

- Flyweights
- Singleton