ABOUT ROHIT JOSHI

Rohit Joshi is a Senior Software Engineer from India. He is a Sun Certified Java Programmer and has worked on projects related to different domains. His expertise is in Core Java and J2EE technologies, but he also has good experience with front-end technologies like Javascript, JQuery, HTML5, and JQWidgets.

State Design Pattern

Posted by: Rohit Joshi in Core Java September 30th, 2015

This article is part of our Academy Course titled Java Design Patterns.

In this course you will delve into a vast number of Design Patterns and see how those are implemented and utilized in Java. You will understand the reasons why patterns are so important and learn when and how to apply each one of them. Check it out here!

Table Of Contents

1. Introduction

To illustrate the use of the State Design Pattern, let us help a company which is looking to build a robot for cooking. The company wants a simple robot that can simply walk and cook. A user can operate a robot using a set of commands via remote control. Currently, a robot can do three things, it can walk, cook, or can be switched off.

The company has set protocols to define the functionality of the robot. If a robot is in "on" state you can command it to walk. If asked to cook, the state would change to "cook" or if set to "off", it will be switched off.

Similarly, when in "cook" state it can walk or cook, but cannot be switched off. And finally, when in "off" state it will automatically get on and walk when the user commands it to walk but cannot cook in off state.

This might look like an easy implementation: a robot class with a set of methods like walk, cook, off, and states like on, cook, and off. We can use if-else branches or switch to implement the protocols set by the company. But too much if-else or switch statements will create a

maintenance nightmare as complexity might increase in the future.

You might think that we can implement this without issues using if-else statements, but as a change comes the code would become more complex. The requirement clearly shows that the behavior of an object is truly based on the state of that object. We can use the State Design Pattern which encapsulates the states of the object into another individual class and keeps the context class independent of any state change.

Let's first know about the State Design Pattern and then we will implement it to solve the problem above.

2. What is the State Design Pattern

The State Design Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

The state of an object can be defined as its exact condition at any given point of time, depending on the values of its properties or attributes. The set of methods implemented by a class constitutes the behavior of its instances. Whenever there is a change in the values of its attributes, we say that the state of an object has changed.

The State pattern is useful in designing an efficient structure for a class, a typical instance of which can exist in many different states and exhibit different behavior depending on the state it is in. In other words, in the case of an object of such a class, some or all of its behavior is completely influenced by its current state. In the State design pattern terminology, such a class is referred to as a

```
Context
```

class. A

```
Context
```

object can alter its behavior when there is a change in its internal state and is also referred as a Stateful object.

The State pattern suggests moving the state-specific behavior out of the

```
Context
```

class into a set of separate classes referred to as

```
State
```

classes. Each of the many different states that a

```
Context
```

object can exist in can be mapped into a separate

```
State
```

class. The implementation of a State class contains the context behavior that is specific to a given state, not the overall behavior of the context itself. The context acts as a client to the set of State objects in the sense that it makes use of different State objects to offer the necessary state-specific behavior to an application object that uses the context in a seamless manner.

By encapsulating the state-specific behavior in separate classes, the context implementation becomes simpler to read: free of too many conditional statements such as if-else or switch-case constructs. When a

```
Context
```

object is first created, it initializes itself with its initial State object. This State object becomes the current State object for the context. By replacing the current State object with a new State object, the context transitions to a new state.

The client application using the context is not responsible for specifying the current State object for the context, but instead, each of the State classes representing specific states are expected to provide the necessary implementation to transition the context into other states. When an application object makes a call to a

```
Context
```

method (behavior),it forwards the method call to its current State object.

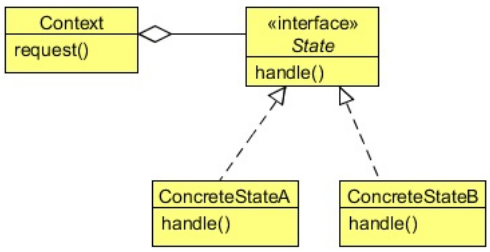

Figure 1 – Class Diagram

Context

- Defines the interface of interest to clients.
- Maintains an instance of a

```
ConcreteState
```

subclass that defines the current state.

State

- Defines an interface for encapsulating the behavior associated with a particular state of the Context.

ConcreteState subclasses

- Each subclass implements a behavior associated with a state of the `Context`

.

3. Implementing the State Design Pattern

The following is the `RoboticState` interface which contains the behavior of a robot.

```
1  package com.javacodegeeks.patterns.statepattern;
2
3  public interface RoboticState {
4
5      public void walk();
6      public void cook();
7      public void off();
8
9  }
```

The `Robot` class is a concrete class implements the `RoboticState` interface. The class contains the set of all possible states a robot can be in.

```
01 package com.javacodegeeks.patterns.statepattern;
02
03 public class Robot implements RoboticState{
04
05     private RoboticState roboticOn;
06     private RoboticState roboticCook;
07     private RoboticState roboticOff;
08
09     private RoboticState state;
10
11     public Robot(){
12         this.roboticOn = new RoboticOn(this);
13         this.roboticCook = new RoboticCook(this);
14         this.roboticOff = new RoboticOff(this);
15
16         this.state = roboticOn;
17     }
18
19     public void setRoboticState(RoboticState state){
20         this.state = state;
21     }
22
23     @Override
24     public void walk() {
25         state.walk();
26     }
27
28     @Override
29     public void cook() {
30         state.cook();
31
32     }
33
34     @Override
35     public void off() {
36         state.off();
37     }
38
39     public RoboticState getRoboticOn() {
40         return roboticOn;
41     }
42
43     public void setRoboticOn(RoboticState roboticOn) {
44         this.roboticOn = roboticOn;
45     }
46
47     public RoboticState getRoboticCook() {
48         return roboticCook;
49     }
50
51     public void setRoboticCook(RoboticState roboticCook) {
52         this.roboticCook = roboticCook;
```

```
    }

    public RoboticState getRoboticOff() {
        return roboticOff;
    }

    public void setRoboticOff(RoboticState roboticOff) {
        this.roboticOff = roboticOff;
    }

    public RoboticState getState() {
        return state;
    }

    public void setState(RoboticState state) {
        this.state = state;
    }

}
```

The class initializes all the states and sets the current state as on.

Now, we will see all the concrete states of a robot. A robot will be in any of these states at any time.

```
package com.javacodegeeks.patterns.statepattern;

public class RoboticOn implements RoboticState{

    private final Robot robot;

    public RoboticOn(Robot robot){
        this.robot = robot;
    }

    @Override
    public void walk() {
        System.out.println("Walking...");
    }

    @Override
    public void cook() {
        System.out.println("Cooking...");
        robot.setRoboticState(robot.getRoboticCook());
    }

    @Override
    public void off() {
        robot.setState(robot.getRoboticOff());
        System.out.println("Robot is switched off");

    }

}
```

```
package com.javacodegeeks.patterns.statepattern;

public class RoboticCook implements RoboticState{

    private final Robot robot;

    public RoboticCook(Robot robot){
        this.robot = robot;
    }

    @Override
    public void walk() {
        System.out.println("Walking...");
        robot.setRoboticState(robot.getRoboticOn());
    }

    @Override
    public void cook() {
        System.out.println("Cooking...");
    }

    @Override
    public void off() {
        System.out.println("Cannot switched off while cooking...");
    }
}
```

```
package com.javacodegeeks.patterns.statepattern;

public class RoboticOff implements RoboticState{

    private final Robot robot;

    public RoboticOff(Robot robot){
        this.robot = robot;
    }

    @Override
    public void walk() {
        System.out.println("Walking...");
        robot.setRoboticState(robot.getRoboticOn());
    }

    @Override
    public void cook() {
        System.out.println("Cannot cook at Off state.");
    }

    @Override
```