State Design Pattern in Java

Introduction

State design pattern is one of the behavioral design pattern. State design pattern is used when an Object change its behavior based on its internal state.

If we have to change the behavior of an object based on its state, we can have a state variable in the Object. Then use if-else condition block to perform different actions based on the state. State design pattern is used to provide a systematic and loosely coupled way to achieve this through Context and State implementations.

State Pattern Context is the class that has a State reference to one of the concrete implementations of the State. Context forwards the request to the state object for processing. Let's understand this with a simple example.

Real time use case

Suppose we want to implement a TV Remote with a simple button to perform action. If the State is ON, it will turn on the TV and if state is OFF, it will turn off the TV.

We can implement it using if-else condition like below.

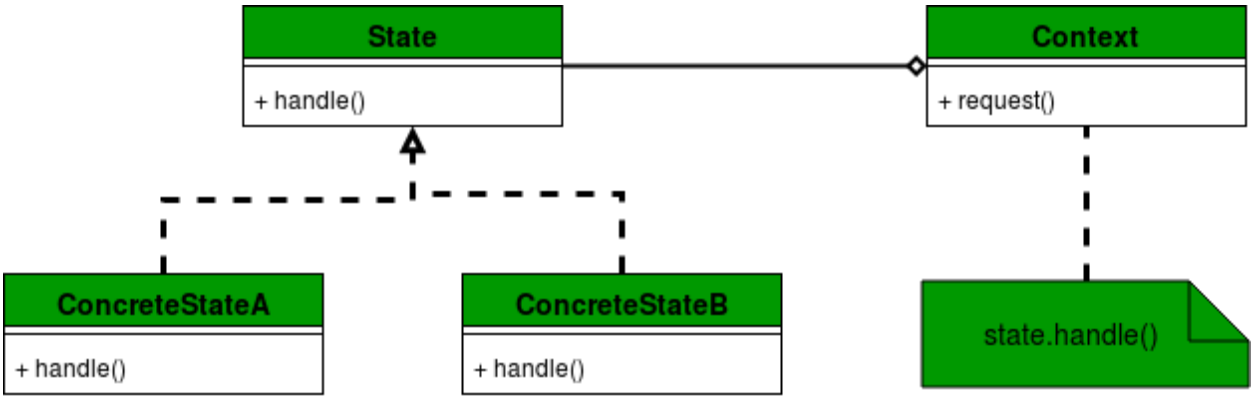
```
package com.journaldev.design.state;

public class TVRemoteBasic {

	private String state="";

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

	public void doAction(){
		if(state.equalsIgnoreCase("ON")){
			System.out.println("TV is turned ON");
		}else if(state.equalsIgnoreCase("OFF")){
			System.out.println("TV is turned OFF");
		}
	}
```

```
    public static void main(String args[])
    {
            TVRemoteBasic remote = new TVRemoteBasic();

            remote.setState("ON");
            remote.doAction();

            remote.setState("OFF");
            remote.doAction();
     }

}
```

Notice that client code should know the specific values to use for setting the state of remote. Further more if number of states increase then the tight coupling between implementation and the client code will be very hard to maintain and extend.

Now we will use State pattern to implement above TV Remote example.

State Design Pattern Interface



First of all we will create State interface that will define the method that should be implemented by different concrete states and context class.

```
public interface State {

        public void doAction();
}
```

State Design Pattern Concrete State Implementations

In our example, we can have two states – one for turning TV on and another to turn it off. So we will create two concrete state implementations for these behaviors.

```
public class TVStartState implements State {

	@Override
	public void doAction() {
		System.out.println("TV is turned ON");
	}

}
```

```
public class TVStopState implements State {

	@Override
	public void doAction() {
		System.out.println("TV is turned OFF");
	}

}
```

Now we are ready to implement our Context object that will change its behavior based on its internal state.

```
public class TVContext implements State {

	private State tvState;

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

	public State getState() {
		return this.tvState;
	}

	@Override
	public void doAction() {
		this.tvState.doAction();
	}
}
```

Notice that Context also implements State and keep a reference of its current state and forwards the request to the state implementation.

A simple program to test our state pattern implementation of TV Remote.

```
public class TVRemote {

	public static void main(String[] args) {
		TVContext context = new TVContext();
		State tvStartState = new TVStartState();
		State tvStopState = new TVStopState();

		context.setState(tvStartState);
		context.doAction();
		context.setState(tvStopState);
		context.doAction();

	}

}
```

State Design Pattern Benefits

The benefits of using State pattern to implement polymorphic behavior is clearly visible. The chances of error are less and it's very easy to add more states for additional behavior. Thus making our code more robust, easily maintainable and flexible. Also State pattern helped in avoiding if-else or switch-case conditional logic in this scenario.