State Design Pattern in Java

APRIL 2, 2018 BY PANKAJ — 10 COMMENTS

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.

Table of Contents [hide]

State Design Pattern

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.

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;

`TVRemoteBasic.java`

```java
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();
```

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.

`State.java`

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

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.

`TVStartState.java`

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

public class TVStartState implements State {

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

}
```

`TVStopState.java`

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

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.

State Design Pattern Context Implementation

`TVContext.java`

```
package com.journaldev.design.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.

State Design Pattern Test Program

Now let's write a simple program to test our state pattern implementation of TV Remote.

`TVRemote.java`

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

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();

        }

}
```

Output of above program is same as the basic implementation of TV Remote without using state pattern.

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.

State Pattern is very similar to Strategy Pattern, check out **Strategy Pattern in Java**.

Thats all for State design pattern in java, I hope you liked it.

« PREVIOUS
Observer Design Pattern in Java

NEXT »
Strategy Design Pattern in Java – Example Tutorial

About Pankaj
If you have come this far, it means that you liked what you are reading. Why not reach little more and connect with me directly on **Google Plus**, **Facebook** or **Twitter**. I would love to hear your thoughts and opinions on my articles directly.

Recently I started creating video tutorials too, so do check out my videos on **Youtube**.

FILED UNDER: DESIGN PATTERNS

Comments

DrJ says
MAY 20, 2015 AT 1:14 AM

Please change all the references of "it's" to "its". They're driving me crazy (as I work on my dissertation I am sensitive to these things).
Reply

Lalit Upadheyay says
DECEMBER 25, 2014 AT 4:15 AM
why TVContext needs to implement State interface ? I think its not required.
Reply

Mednnet says
JULY 12, 2014 AT 5:37 PM
According to me, this is an anti-pattern. This is like having a method whith a boolean parameter and 2 actions in it, depending on this boolean.
It doesn't respect the seperation of concerns and the single responsability principle : 1 method for 1 action.
If i need to do the action 1 in a case and the action 2 in another case i'll create action 1 and action2 méthods.
With the same example :
method 1 : public boolean isTheTvOn();
method 2 : public void turnOnTheTv();
method 3 : public void turnOffTheTv();
and determine the action to do in a service layer
Reply

Hector Fontanez says
JULY 15, 2014 AT 6:58 PM
You are correct in the sense that the pattern was not implemented correctly. The context class should use the stored state object to call the doWhatever() method and not go to a series of nested ifs to figure out what to do. In my opinion, it is also better to show this example with three states instead of two. Toggling between two states is too simplistic. So:

```
public abstract class State
{
public abstract next(Context c);
public abstract previous(Context c);
}
```

The concrete state classes must define what are the valid transitions from a current state (if any). Applying this to a simple navigation (wizard), you don't want to allow navigation from the first state to

the third and from the third to the first. Only the second state should allow both a previous and next states.
The context class should just use the stored (current) state to determine what to do:

```
public class Context
{
private State current;
...
public void setCurrent(State state)
{
current = state;
}
public void goNext()
{
current.next(this);
}
public void goPrev()
{
current.prev(this);
}
}
```

A possible scenario for a concrete state goes as follows:

```
public final class SecondState extends State
{
private static SecondState INSTANCE = new SecondState();
private SecondState() {}
@Override
public void next(Context context)
{
context.setCurrent(ThirdState.getInstance());
System.out.println( "Second to Third state transition" );
}
@Override
public void previous(Context context)
{
context.setCurrent(FirstState.getInstance());
System.out.println( "Second to First state transition" );
}
}
```

This respects both the separation of concerns and the single responsibility principle.
Reply

Prasanti says

APRIL 2, 2018 AT 9:03 PM

Thanks for posting this.

Reply

Someone Better says

MAY 6, 2016 AT 5:39 AM

Unfortunately the explanation is wrong. The client in pattern implementation version also sets the required implementation object, that is not how state pattern works. Blogs like this produce huge content with diluted concepts, which is very bad.

Reply

JavaAmateur says

APRIL 2, 2018 AT 9:08 PM

Yes, I am looking for a standard book that has valid and correct explanations (like SCJP by Kathy Sierra and Bert Bates) of design patterns. If you are aware of any, could you please suggest?

Reply

Nisha says

MAY 15, 2014 AT 10:05 AM

Very good example to understand the state pattern implementation

Reply

Adelin says

APRIL 23, 2014 AT 1:17 AM

Hi and thanks for the nice post.
The thing that I didn't understand is what is the purpose of the context its only job is to call doAction() of the provided state, why don't client code call doAction() of state instead, why do I need to create a Context that its only job is to call doAction() ?
Thanks

Reply

Neal says

MARCH 27, 2014 AT 6:31 PM

Great explanation, but it's little bit confusing where it changes the object alternative.

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

Search for tutorials...

DOWNLOAD ANDROID APP

GET IT ON Google Play

DESIGN PATTERNS TUTORIAL

RECOMMENDED TUTORIALS

- Java Generics
- Exception Handling in Java
- Java Reflection
- Java Design Patterns
- JDBC Tutorial

Java EE Tutorials

- Servlet JSP Tutorial
- Struts2 Tutorial
- Spring Tutorial
- Hibernate Tutorial
- Primefaces Tutorial
- Apache Axis 2
- JAX-RS
- Memcached Tutorial

Enforce updates on *private android* apps

SIGN UP FREE

pushlink