

The State Pattern encapsulates state expressions in objects. This can be useful when an object shows different behavior depending on its state. Think of a garage door. The door can be open, but it can also be closed. You can open a closed door; however, there is little point in trying to open an open door. But how can you prevent someone from trying to open an open gate or close a closed gate? The state pattern solves this problem by representing each state by its own object; as a result, everyone can only do what is supposed to be allowed in the current state.

8.1 Excursus: The Enum Pattern

Let's first digress and look at the Enum Pattern before we tackle the State Pattern.

8.1.1 Representing a State by Numerical Values

If you want to represent two state expressions, you quickly get the idea of representing these state expressions by numerical values and passing these values to final variables. Within a switch statement you query the values and react to them:

```
public class GateManager_1 {  
    static final int OPEN = 0;  
    static final int CLOSED = 1;  
    private int state = OPEN;  
  
    void printState() {
```

```

        switch (state) {
            case OPEN ->
                System.out.println("Gate is open");
            case CLOSED ->
                System.out.println("Gate is closed");
            default ->
                System.out.println("**** Error ****");
        }
    }

    void setState(int state) {
        this.state = state;
    }

    // ... abridged
}

```

In this code section, pay attention to the Switch Statement(!), where I use the arrow syntax after the respective case, which was inserted with Switch Expressions (compare my first explanations of Switch Expressions in Sect. 5.7). Switch statements also benefit from this syntax innovation, because I save the `break` commands here.

When you create an object of this class, you can either pass one of the defined constants to the `setState()` method, or you can pass an arbitrary `int` number by error. Depending on the state, a different text is output to the console in each case. In the example project `GateManager` you will find the classes `GateManager_1` to `GateManager_5` described above or in the following. For the sake of simplicity, I have combined them into one NetBeans project. In `GateManager_1` you will find a main method within the class that demonstrates the procedure described above.

8.1.2 Representing a State by Objects

An alternative approach is to represent the state characteristics by objects of their own data type. To do this, declare a class `State`. Within this class, two static final variables of this type are created. To prevent the user from creating further objects of the type `state`, the constructor is declared private – access is now only possible within the class. The technique is the same as you have already seen with the Singleton pattern. You can find said data type as a static inner class in the `GateManager_2` class.

```

public class GateManager_2 {
    // ... abridged
    public static class State {
        public static final state OPEN =
            new state("open");
    }
}

```

```
public static final state CLOSED =
    new State("closed");

private final String description;

private state(string description) {
    this.description = description;
}

// ... abridged
private state state = state.OPEN;

void setState(State state) {
    if (state != null)
        this.state = state;
}
// ... abridged
}
```

Now, only the state characteristics specified in the `State` class can be used. So this solution has become type safe. Since this approach was used very often, it was called Enum Pattern. One disadvantage is that you can no longer query the state expressions in a switch statement during processing.

8.1.3 Implementation in the Java Class Library

Programmers are often faced with the necessity of representing state expressions in a type-safe manner. This is what enumerations are for in Java. The compiler translates enums so that the bytecode corresponds exactly to the enum pattern shown above. Instead of `class`, you now write `enum`. Also, a bit of typing has been taken away from the programmer; the modifiers `static final` and the explicit object creation can be omitted; the various options are separated by commas. You can find this variant in the class `GateManager_3`.

```
enum state {
    OPEN("open"), CLOSED("closed");
    private final String description;

    private state(string description) {
        this.description = description;
    }
}
```

Enums again allow you to query state expressions in a switch statement:

```
void printState() {  
    switch (state) {  
        case OPEN ->  
            System.out.println("Gate is open");  
        case CLOSED ->  
            System.out.println("Gate is closed");  
    }  
}
```

Are you missing the default case in this switch statement? Since the checked state is an enum datatype, we don't need that anymore as long as all the individual values are also covered in the cases. That is the case here, and thus the need for the default case is eliminated. Another handy Java customization introduced with Switch Expressions and also available for Switch Statements.

In the next section, you will change the states of an object.

8.2 Changing the State of an Object

In the previous paragraph, you saw how you can define state expressions in a type-safe way. It can now be interesting to see how you transfer one state to another and how the behavior of an object is changed as a result.

8.2.1 A First Approach

In the `GateManager_4` class, the first step is to consider how to transition one state to another:

```
public class GateManager_4 {  
    enum state {  
        // ... abridged  
    }  
  
    private state state = state.OPEN;  
  
    private void open() {  
        System.out.println("The gate is opened");  
        this.state = state.OPEN;  
    }  
}
```

```
private void close() {
    System.out.println("The gate is closed");
    this.state = state.CLOSED;
}
}
```

Actually, this solution is optimal, isn't it? When a gate is closed, you open it to transfer it to the other state. If it is open, you close it and then also have a different state. But with this coding, it is possible for the user to open a gate that is open or close one that is closed. And that's exactly what shouldn't happen. So query the state expressions beforehand!

8.2.2 A Second Approach

Include an if statement in each method that allows opening or closing only if it changes the object to a different state. If the user tries to open an open gate or close a closed one, alert them with an error message. You can find this code in the class `GateManager_5`.

```
public class GateManager_5 {
    enum state {
        // ... abridged
    }
    private state state = state.OPEN;

    private void open() {
        if (state == state.CLOSED) {
            System.out.println("The gate is opened");
            this.state = state.OPEN;
        }
        else
            System.out.println("Gate is already open");
    }

    private void close() {
        if (state == state.OPEN) {
            System.out.println("The gate is closed");
            this.state = state.CLOSED;
        }
        else
            System.out.println("Gate is already closed");
    }
}
```

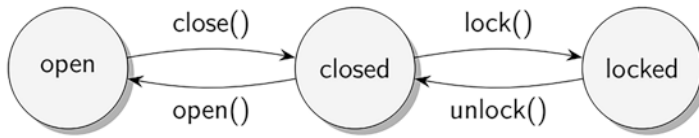


Fig. 8.1 Three state characteristics and their transitions

This solution works flawlessly. But where could it show weaknesses? It is no longer useful when further state expressions are added and there are extensive dependencies between them.

I would like to extend the example by the fact that your client requests a new condition. The gate should not only be closed, but also be able to be locked with a lock. You now have the third state LOCKED. The state LOCKED may only be set if the gate is closed. Conversely, the LOCKED state can only be changed to the CLOSED state. Figure 8.1 shows the three states and their possible transitions.

You need to relate three state expressions and make the object's behavior dependent on them. When a fourth state is added, you can imagine the work that will come your way.

How the method `open()` could look like in this case is shown in the following listing, which is only commented out in the sample code:

```

private void open() {
    switch (state) {
        case OPEN ->
            System.out.println("Gate is already open");
        case CLOSED -> {
            System.out.println("The gate is opened");
            this.state = state.OPEN;
        }
        case LOCKED ->
            System.out.println("The gate is locked");
    }
};
}

```

Similarly, the other four methods would have to define the correct behavior for each state – even if the behavior is only to issue an appropriate error message. The scope of the methods increases with each new state. The class becomes unwieldy and difficult to maintain. Errors can creep in very quickly. In short: It's high time for the State Pattern.

8.3 The Principle of the State Pattern

The definition of the state characteristics by enumerations is no longer sufficient. Describe each state by its own class.

8.3.1 Defining the Role of All States

In order to be able to exchange the state expressions, they must have a common data type. You first create the abstract class `State`, in which all methods from the state diagram are provided. In addition, this class holds a reference to the goal, which is passed to it in the constructor. The following code can be found in the `StatePattern_1` sample project.

```
public abstract class State {  
    public final Gate gate;  
    State(Gate gate) {  
        this.gate = gate;  
    }  
    abstract void open();  
    abstract void close();  
    abstract void lock();  
    abstract void unlock();  
}
```

Each state is represented by its own class and extends the abstract class. So, within the state, you define what should happen when a particular method, for example `open()`, is called. When you define a state, you need to think about what a method in that state should look like. Let's walk through the example using the Open state. If a gate is open and someone tries to open it again, print an error message. Similarly, an open gate cannot be unlocked or locked; therefore, print error messages in these methods as well. However, if the user wants to close an open gate, change the state to closed.

```
public class Open extends State {  
    Open(Gate gate) {  
        super(gate);  
    }  
  
    @Override  
    public void open() {  
        System.out.println("The gate is already open.");  
    }  
  
    @Override  
    public void close() {  
        System.out.println("The gate will be closed.");  
        gate.setState(new Closed(tor));  
    }  
  
    @Override  
    public void lock() {
```

```

        System.out.println("Close the gate first.");
    }

    @Override
    public void unlock() {
        System.out.println("The gate is not locked.");
    }
}

```

The other state classes are defined accordingly.

How does this change affect the gate? The Gate class defines a state field that references the current state. All the methods you know from the state diagram can be called on this state. So the gate itself can define an `open()` method that passes the call to the state field. Here I print the abbreviated source code of the Gate class.

```

public class Gate {
    private State state = new Open(this);

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

    public void open() {
        state.open();
    }

    // ... abridged
}

```

The gate calls the desired method on the current state. However, it has no idea which state is currently stored. If you call the same method twice on the same object, a different state may be active, the behavior of the object may be different, and you may have the impression that you are working with the instance of a different class.

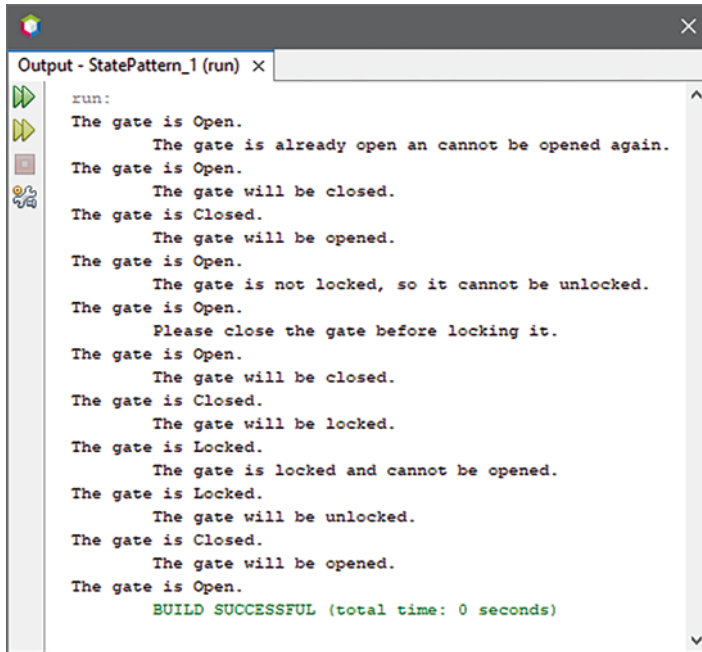
8.3.2 The Project from the Client's Point of View

In the client class of the project, the Gate is put to use. Compare the source code, which I only print here in abbreviated form, with the console output:

```

public static void main(String[] args) {
    GATE.open();
    GATE.close();
    GATE.open();
}

```

```

run:
The gate is Open.
    The gate is already open an cannot be opened again.
The gate is Open.
    The gate will be closed.
The gate is Closed.
    The gate will be opened.
The gate is Open.
    The gate is not locked, so it cannot be unlocked.
The gate is Open.
    Please close the gate before locking it.
The gate is Open.
    The gate will be closed.
The gate is Closed.
    The gate will be locked.
The gate is Locked.
    The gate is locked and cannot be opened.
The gate is Locked.
    The gate will be unlocked.
The gate is Closed.
    The gate will be opened.
The gate is Open.
BUILD SUCCESSFUL (total time: 0 seconds)

```

Fig. 8.2 Console output of the sample project StatePattern_1

```

GATE.unlock();
GATE.lock();
GATE.close();
GATE.lock();
GATE.open();
GATE.unlock();
GATE.open();
}

```

The resulting console output can be found in Fig. 8.2.

Notice how many times the `open()` method is called: four times in total. With each call, the context, i.e. the gate, reacts differently. For the client, this solution is very convenient – it does not know how the gate works internally, it especially does not have to deal with the different state classes.

Nevertheless, two disadvantages should not be concealed. First, you need a separate class for each state. Some state characteristics depend “somehow” on each other, others not at all. This creates the risk that the project becomes confusing. Without a clear state diagram, you may be at a loss.

A second point can become a problem. You create a new object with every state change. If you have frequent state changes or the state class is very costly to instantiate, this solution is not to be favored. Let’s look at alternative implementations below.

8.3.3 Changes to the Project

I would like to present two options. The first alternative is to manage the state objects centrally in the context. The second alternative is to create the state objects in the superclass of all state classes and to pass the correct object to the caller with each method call.

8.3.3.1 Manage State Objects Centrally in Context

If you don't want to create a new object every time the state changes, you need an alternative solution. Let the context define variables that represent each state. Also, for each state, the context defines a method that sets the State `currentState` field to the desired state. You can find the code for this in the `StatePattern_2` sample project:

```
public class Gate {
    private final state openState = new Open(this);
    private final state closedState = new Closed(this);
    private final state lockedState = new Locked(this);

    private state currentState = openState;

    public void setOpenState() {
        currentState = openState;
    }

    public void open() {
        currentState.open();
    }

    // ... abridged
}
```

The state classes now no longer pass state objects to the context, but instruct the context to assume a certain state. How it does that is up to it. However, this binds the context very strongly to the state classes.

8.3.3.2 State Objects as Return Values of Method Calls

Another alternative implementation is that you basically return state objects as return values. If the state does not change, it returns a reference to itself, that is, `this`. To do this, you define variables in the abstract superclass that represent the various state expressions. You can find this variant in the sample project `StatePattern_3`:

```
public abstract class State {
    protected static final state OPEN = new Open();
    protected static final state CLOSED = new Closed();
```

```
protected static final state LOCKED = new Locked();

abstract state open();

// ... abridged
}
```

The final variables are declared `protected` to allow the subclasses to access them.

```
public class Open extends State {
    @Override
    public State open() {
        System.out.println("The gate is already open.");
        return this;
    }

    @Override
    public State close() {
        System.out.println("The gate is closed.");
        return super.CLOSED;
    }
    // ... abridged
}
```

The context is now responsible for setting the received state object as the current state. This makes the project very flexible, new state classes can be inserted easily. Because the context no longer has to provide a set method, incorrect state values are not accidentally passed or set.

8.4 The State Pattern in Practice

You will find the state pattern quite often in practice. For example, network protocols often work with state expressions. As an example, I have chosen RFC 1939, which describes POP3. You can find the text of the RFC in the subfolder for the state pattern.

RFC 1939 recognizes three different state specifications: AUTHORIZATION, TRANSACTION, and UPDATE. These are actually referred to as states in the RFC. The POP3 server waits for requests. When a request arrives, a TCP connection is established. The session is initially in the AUTHORIZATION state, where the user's name and password are requested. If the name can be identified and the password is correct, the session enters the TRANSACTION state. In this state, commands can be sent to the server. For example, a list of stored e-mails can be requested. Furthermore, an e-mail can be marked for deletion. When the QUIT command is received, the session changes to the UPDATE state. In this state, the e-mails marked for deletion are actually deleted. Afterwards, the connection is terminated.

If the QUIT command is issued in the AUTHORIZATION state, the session is terminated without UPDATE. Calling QUIT therefore has two completely different effects depending on the state. Commands that refer to the stored messages are only permitted in the TRANSACTION state.

Take a look at the sample POP3 project, which demonstrates the transition of these three state expressions. I have stored four state classes in the `states` package: Three for the states defined in the protocol and one class for the START state, which is needed when the server is waiting for requests. There is also a class `POP3_Session` in this package. This class is the context that the clients access. The client only “sees” the context. It has no information that a variety of state classes are needed in the background. From this point of view, analyse the `main` method of the `ApplStart` class.

8.5 State – The UML Diagram

The UML-diagram for State-Pattern from the example project `StatePattern_3` can be seen in Fig. 8.3.

8.6 Summary

Go through the chapter again in key words:

The enum pattern allows you to define different state specifications in a type-safe manner.

- Usually a private constructor is defined.
- Instances of the classes are declared static and final.
- The enum pattern is implemented by the enumeration since Java 5.

The most important points about the State Pattern:

- An object can have different state characteristics.
- Depending on the state, the object shows different behavior.
- The transition from one state to another should be specified in the state classes themselves.
- Enumeration is usually not sufficient for this.
- To the client, it looks like it’s dealing with an instance of another class.
- The state instances can either be regenerated each time the state changes or kept in a central location. Which solution you choose depends on whether the state is likely to change frequently or whether the creation of an instance of a state class is cost-intensive. In this case, you create the state instances when you start the program and store them in a central location.

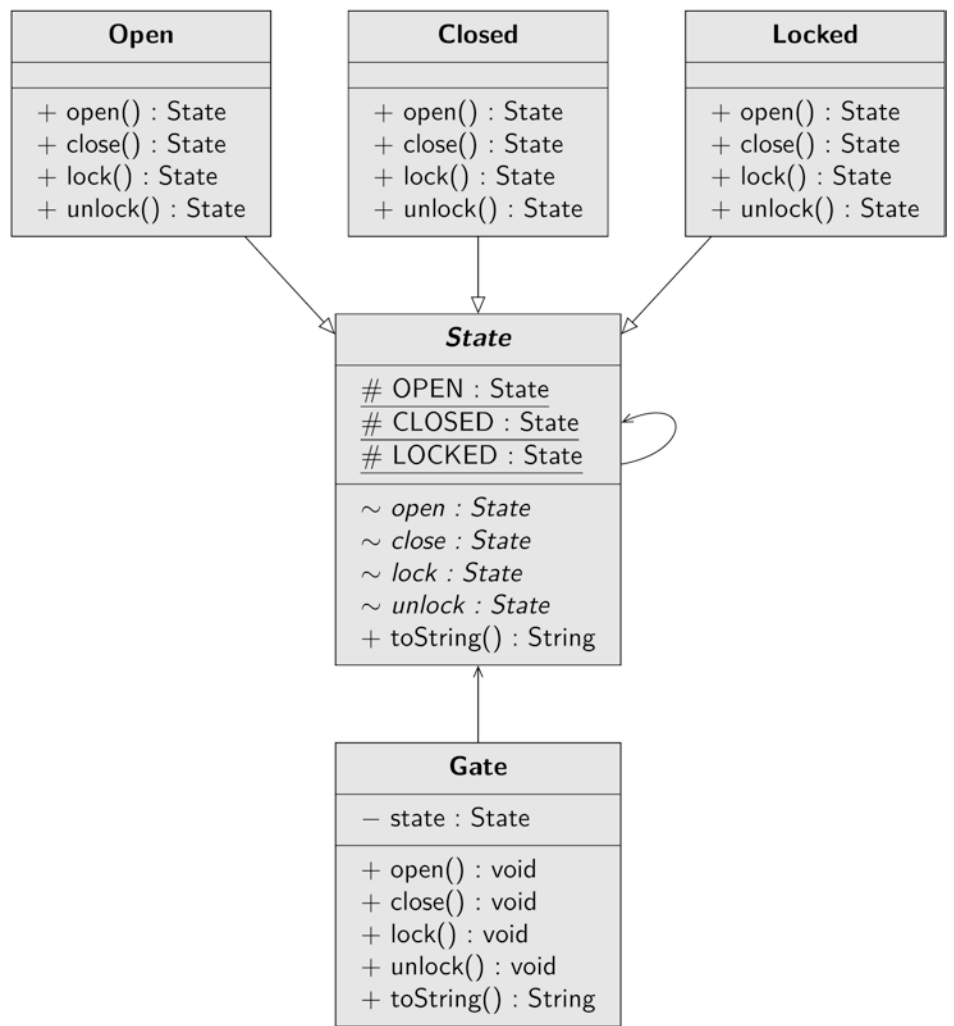


Fig. 8.3 UML diagram of the State Pattern (example project StatePattern_3)

8.7 Description of Purpose

The Gang of Four describes the purpose of the pattern “State” as follows:

Allow an object to change its behavior when its internal state changes. It will look like the object has changed its class.