**State Design Pattern in Java**

The State Design Pattern allows the context (the object that has a certain state) to behave differently based on the currently active ConcreteState instance



Let’s take a closer look into the parts that make up the state design pattern.

#### Context Object

Context is an instance of a class that owns (contains) the state. The context is an object that represents a thing that can have more than one state. In fact, it could have many different states. There is really no limit. It is perfectly fine to have many possible state objects even into the hundreds. It is coming to have context objects with only a handful of possible states, though.

The Context object has at least one method to process requests and passes these requests along to the state objects for processing. The context has no clue on what the possible states are. The context must not be aware of the meaning of these different states. It is important that the context object does not do any manipulation of the states (no state changes). The only exception is that the context may set an initial state at startup and therefore must be aware of the existence of that initial state. This initial state can be set in code or come from an external configuration.

The only concern that the context has is to pass the request to the underlying state object for processing. The big advantage of not knowing what states the context could be in is that you can add as many new states as required over time. This makes maintaining the context super simple and super flexible. A true time saver and a step closer to being rich beyond your wildest dreams (almost).

#### State

The State class is an abstract class. It is usually an abstract class and not an interface (IInterface). This class is the base class for all possible states. The reason why this class is usually an abstract class and not an interface is because there are usually common actions required to apply to all states. These global methods can be implemented in this base class. Since you can’t do any implementation in Interfaces, abstract classes are perfect for this. Even if you do not have any initial global base methods, use abstract classes anyways because you never know if you might need base methods later on.

The State class defines all possible method signatures that all states must implement. This is extremely important to keep the maintenance of all possible states as simple as possible. Since all states will implement these methods signatures and if you forget to implement a new method, the compiler will warn you at compile time. An awesome safety net.

#### ConcreteState

The ConcreteState object implements the actual state behavior for the context object. It inherits from the base State class. The ConcreteState class must implement all methods from the abstract base class State.

The ConcreteState object has all the business knowledge required to make decisions about its state behavior. It makes decisions on when and how it should switch from one state to another. It has knowledge of other possible ConcreteState objects so that it can switch to another state if required.

The ConcreteState object can even check other context objects and their states to make business decisions. Many times, an object may have more than one context object. When this happens, a ConcreteState object may need to access these different states and make a decision based on active states. This allows for complicated scenarios but fairly easy to implement using the state design pattern. You will see an example later in this article that shows multiple context objects and their states and the need to work together.

The ConcreteState object also is capable of handling before and after transitioning to states. Being aware of a transition about to happen is an extremely powerful feature. For example, this can be used for logging, audit recording, security, firing off external services, kicking of workflows, etc. and many other purposes.

# State

Also known as Objects for States

## **Pattern Properties**

Type: Behavioral

Level: Object

## **Purpose**

To easily change an object’s behavior at runtime.

## **Introduction**

An application often behaves differently depending on the values of its internal variables. For instance, when you're working on a text file, you need to periodically save your work. Most current text editors allow you to save a document only when something has changed in the text. As soon as you save the content the text is considered to be “clean;” the file content is the same as the content currently on display. At this point the Save option is not available as it serves no purpose.

Implementing this decision-making in the individual methods makes the code hard to maintain and read. The result is that these methods contain long if/ else statements. A common tactic is to store the state of an object in a single variable using constants for a value. With this approach the methods normally contain large switch/case statements that are very similar in each method.

Objects are state and behavior; state is kept in its attributes and the behavior is defined in methods. The State pattern allows you to change the behavior of an object dynamically. This dynamic behavior is achieved by delegating all method calls that rely on certain values to a State object. Such a State object is state and behavior as well, so that when you change State objects, you also receive a different behavior. The methods in the specific State classes no longer have to use if/else or switch statements; the State object defines the behavior for one state.

## **Applicability**

Use the State pattern when:

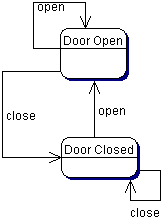
* The object’s behavior depends on its state and the state changes frequently.
* Methods have large conditional statements that depend on the state of the object.

## **Description**

Objects that have different behavior based on their current state might be difficult to implement without the State pattern. As mentioned before, implementation without using the State pattern often results in using constants as a way of keeping track of the current state, and in lengthy switch statements within methods. Most of those methods in the same class have a similar structure (determining the current state).

Consider a door. What are the normal operations you can do with a simple door? You can open and close a door, leaving the door in one of its two states: Closed or Open. Calling the close method on a Closed door accomplishes nothing, but calling the close method on an Open door changes the state of the door to Closed.

The State transition diagram is shown in [Figure 2.11](https://www.safaribooksonline.com/library/view/Applied+Java%E2%84%A2+Patterns/0130935387/ch02.html#ch02fig11).

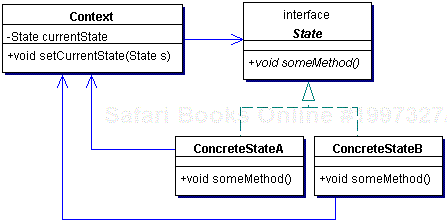


**Figure 2.11. State transition diagram for a door**

The current state of the door makes it behave differently in response to the same command.

## **Implementation**

The class diagram for the State pattern is shown in [Figure 2.12](https://www.safaribooksonline.com/library/view/Applied+Java%E2%84%A2+Patterns/0130935387/ch02.html#ch02fig12).



**Figure 2.12. State class diagram**

Implementing the State pattern requires:

* **Context –.**Keeps a reference to the current state, and is the interface for other clients to use. It delegates all state-specific method calls to the current State object.
* **State –.**Defines all the methods that depend on the state of the object.
* **ConcreteState –.**Implements the State interface, and implements specific behavior for one state.

The Context or the ConcreteState can determine the transition between states. This is not specified by the State pattern. When the number of states is fixed, the most appropriate place to put the transition logic is in the Context.

However, you gain more flexibility by placing the transition logic in the State subclasses. In that case, each State determines the transition—which is the next State, under what circumstances the transition occurs, and when it occurs. This makes it much easier to change part of the Statetransitions and add new States to the system. The drawback is that each class that implementsState is dependent on other classes—each State implementation must know at least one otherState. If the State implementations determine the transition, the Context must provide a way for the State to set the new current State in the Context.

You can create state objects two using two methods: lazy instantiation or upfront creation.

* Lazy instantiation creates the State objects at the time they are needed. This is useful only if the state rarely changes. It is required if the different states are unknown at the start of the application. Lazy instantiation prevents large, costly states from being created if they will never be used.
* Up-front creation is the most common choice. All the state objects are created at startup. You reuse a state object instead of destroying and creating one each time, meaning that instantiation costs are paid only once. This makes sense if the state transitions are frequent—if a state is likely to be needed again soon.

## **Benefits and Drawbacks**

Benefits and drawbacks include the following:

* State partitions behavior based on state – This gives you a much clearer view of the behavior. When the object is in a specific state, look at the corresponding State subclass. All the possible behavior from that state is included there.
* State offers structure and makes its intent clearer – The commonly used alternative to the State pattern is to use constants, and to use a switch statement to determine the appropriate actions. This is a poor solution because it creates duplication. A number of methods use almost exactly the same switch statement structure. If you want to add a new state in such a system you have to change all the methods in the Context class by adding a new element to each switch statement. This is both tedious and error-prone. By contrast, the same change in a system that uses the State pattern is implemented simply by creating one new state implementation.
* State transitions are explicit – When using constants for state, it is easy to confuse a state change with a variable assignment because they are syntactically the same. States are now compartmentalized in objects, making it much easier to recognize a state change.
* State can be shared – If State subclasses contain only behavior and no instance variables, they have effectively become Flyweights. (See “ [Flyweight](https://www.safaribooksonline.com/library/view/Applied+Java%E2%84%A2+Patterns/0130935387/ch03.html#ch03lev1sec6) ” on page 183.) Any state they need can be passed to them by the Context. This reduces the number of objects in the system.
* The State pattern uses a large number of classes – The increased number of classes might be considered a disadvantage. The State pattern creates at least one class for every possible state. But when you consider the alternative (long switch statements in methods), it’s clear that the large number of classes is an advantage, because they present a much clearer view.

## **Pattern Variants**

One of the challenges of the State pattern is determining who governs the state transitions. The choice between the Context and the State subclasses was discussed previously. A third option is to look up the transitions in a table structure, with a table for each state, which maps every possible input to a succeeding state [Car92]. This converts the transition code into a table lookup operation.

The benefit is the regularity. To change the transition criteria, only the data in the table has to be changed instead of the actual code. But the disadvantages are numerous:

* Table lookups are often less efficient than a method call.
* Putting the transition logic in a table makes the logic harder to understand quickly.

The main difference is that the State pattern is focused on modeling the behavior based on the state, whereas the table approach focuses on the transitions between the different states.

A combination of these two approaches combines the dynamics of the table-driven model with the State pattern. Store the transitions in a HashMap, but instead of having a table for each state, create aHashMap for every method in the State interface. That’s because the next state is most likely different for each method.

In the HashMap, use the old state as the key and the new state as the value. Adding a new State is very easy; add the class and have the class change the appropriate HashMaps. This variant is also demonstrated in the Example section for this pattern.

### So How Does It Work In Java?

We'll use the state of an mp3 player to give an example of the state pattern in action. First we set up a context for our mp3 playe.

//Context

public class MP3PlayerContext {

private State state;

private MP3PlayerContext(State state) {

this.state= state;

}

public void play() {

state.pressPlay(this);

}

public void setState(State state) {

this.state = state;

}

public State getState() {

return state;

}

}

Now we'll create our state interface. In this example, we've just got a play button.

private interface State {

public void pressPlay(MP3PlayerContext context);

}

And finally, creating a state for Standby and for Playing.

public class StandbyState implements State {

public void pressPlay(MP3PlayerContext context) {

context.setState(new PlayingState());

}

}

public class PlayingState implements State {

public void pressPlay(MP3PlayerContext context) {

context.setState(new StandbyState());

}

}

So this shows how the state pattern works at a simple level. Of course, our pressPlay methods would do more than simply set the state of the context.