State Behavioral Design Pattern

The biggest problem with state machine id is that it contains so many conditionals and it becomes difficult to predict all possible states and transition to the design state. To solve this problem "State Pattern" was created.

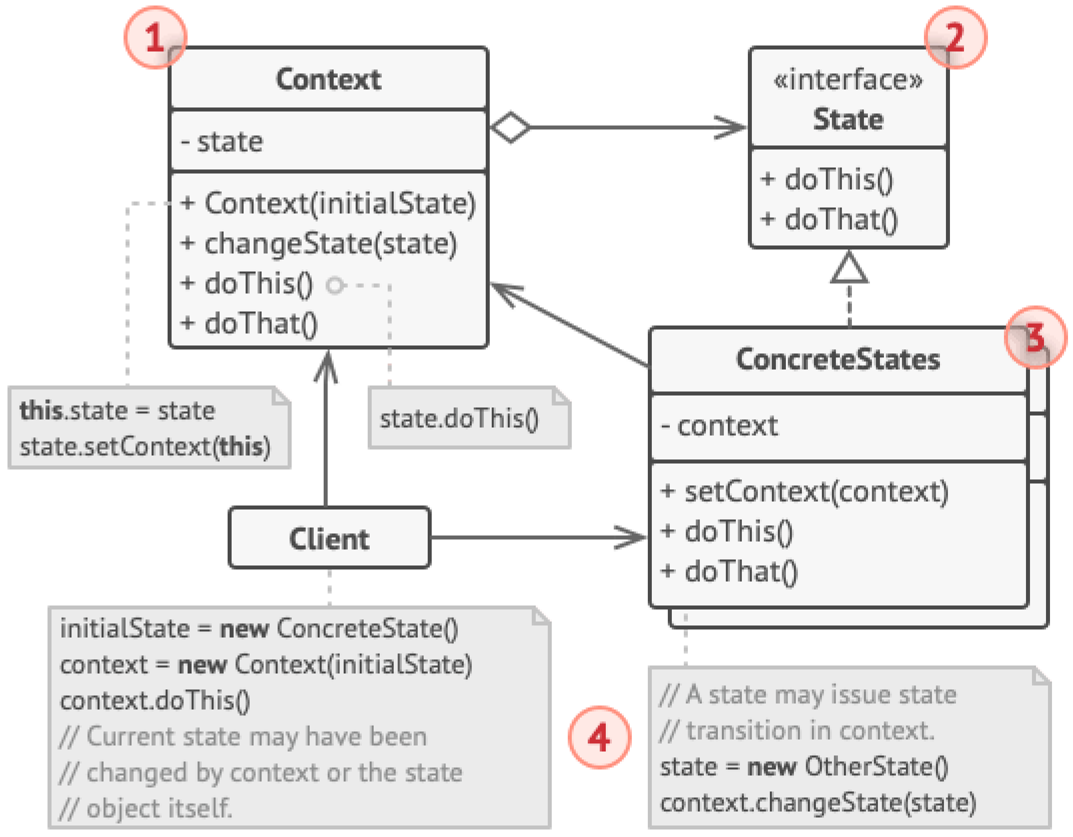
State pattern is a behavioral design pattern that lets an object alter its behavior when its internal states changes. In State Pattern various objects are created which represent various states and it appears as if the object changed its class.

To implement a behavior the original object stores a reference to one of the state objects that represents its current state and assigns all the state related work to that object only.

USAGE:

- When the behavior of object depends on its state and it should be capable of changing its behavior at runtime as per the new state.
- When the operations have large conditional statements that depend on the state of an object.

STRUCTURE:



1. ***Context:*** Context is the original object which stores a reference to one of the concrete state objects and assigns it all the state-related work. The communication between the state object and the context occurs via the state interface.

2. ***State:*** The state interface declares the state related methods which should make make sense to all the concrete method to avoid unnecessary or useless methods from being called.

3. ***Concrete States:*** Concrete-states provides their own implementations for the state-related methods. In order to avoid ambiguity because of duplicate code across multiple states we can provide an intermediate abstract class that encapsulates the common behavior.

4. Both context and concrete states can set next state of the context and perform state transition by replacing the state object linked to the context.

IMPLEMENTATION:

1. Decide what class will act as context.
2. Declare the state interface.
3. For every actual state, create a class that derives from the state interface. Then extract all code from the context methods related to state into newly created class.
4. In context class, add a reference of the state interface type and a public setter that allows overriding the values of that field.
5. Replace empty conditional with calls to corresponding state object methods in context methods.
6. Create an instance of the state class and pass it to the context in order to switch the state of the context.

Example:

1. Create a *Connection* interface that will provide the connection to the Controller class.

```
public interface Connection {
        public void open();
         public void close();
         public void log();
        public void update();
}
```

2. Create an *Accounting* class that will implement to the Connection interface.

```
public class Accounting implements Connection {
    @Override
    public void open() {
      System.out.println("open database for accounting");
    }
    @Override
    public void close() {
      System.out.println("close the database");
    }
    @Override
    public void log() {
      System.out.println("log activities");
    }
    @Override
    public void update() {
       System.out.println("Accounting has been updated");  }
    }
```

3. Create a *Sales* class that will implement to t he Connection interface.

```
public class Sales implements Connection {
      @Override
       public void open() {
          System.out.println("open database for sales");
       }
       @Override
       public void close() {
          System.out.println("close the database");
       }
       @Override
       public void log() {
          System.out.println("log activities");
       }
       @Override
       public void update() {
           System.out.println("Sales has been updated");
       }
}
```

4. Create a *Management* class that will implement to the Connection interface.

```
public class Management implements Connection {
      @Override
       public void open() {
          System.out.println("open database for Management");
       }
       @Override
       public void close() {
          System.out.println("close the database");
       }
       @Override
       public void log() {
          System.out.println("log activities");
       }
       @Override
       public void update() {
           System.out.println("Management has been updated");
       }
}
```

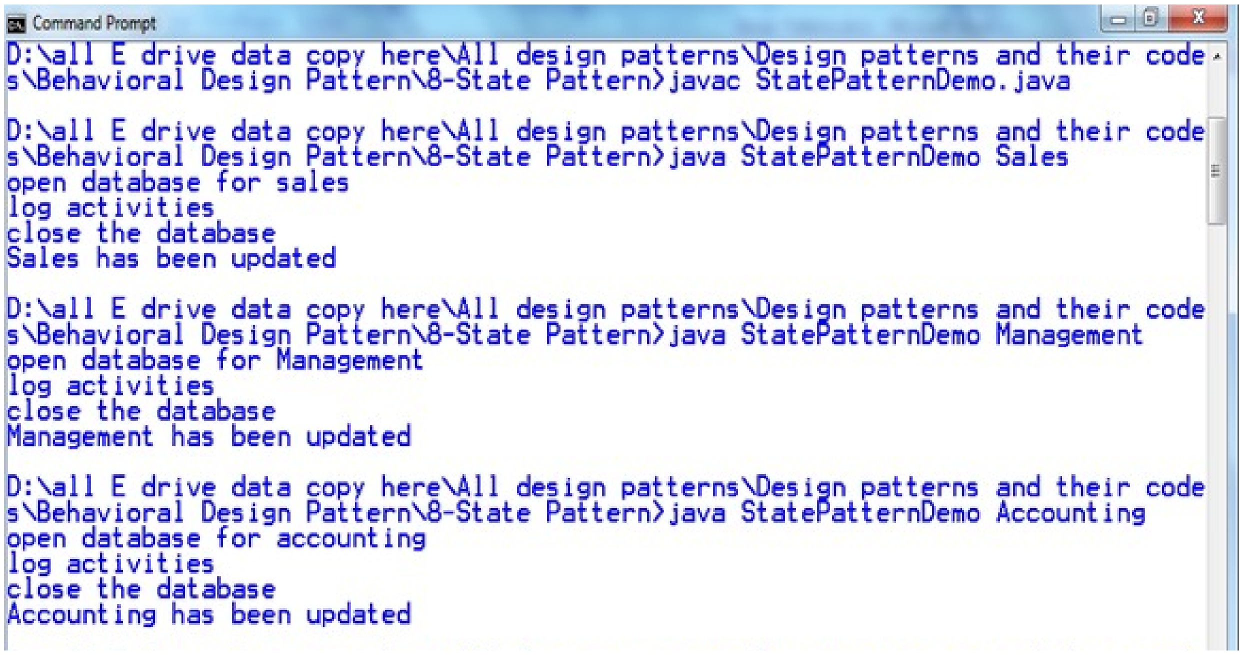
5. Create a *Controller* class that will use the Connection interface for connecting with different types of connection.

```
public class Controller {
       public static Accounting acct;
       public static Sales sales;
       public static Management management;
       private static Connection con;

       Controller() {
           acct = new Accounting();
           sales = new Sales();
           management = new Management();
       }
       public void setAccountingConnection() {
           con = acct;
       }
       public void setSalesConnection() {
           con  = sales;  }
       public void setManagementConnection() {
           con  = management;  }
       public void open() {
           con .open();  }
       public void close() {
           con .close();  }
       public void log() {
           con .log();
       }
       public void update() {
           con .update();
       }
}
```

6. Create a *StatePatternDemo* class.

```
public class StatePatternDemo {
        Controller controller;
        StatePatternDemo(String con) {
           controller = new Controller();
           if(con.equalsIgnoreCase("management"))
              controller.setManagementConnection();
           if(con.equalsIgnoreCase("sales"))
              controller.setSalesConnection();
           if(con.equalsIgnoreCase("accounting"))
                  controller.setAccountingConnection();
           controller.open();
           controller.log();
           controller.close();
           controller.update();
        }
        public static void main(String args[]) {
            new StatePatternDemo(args[0]);  }
}
```



```
D:\all E drive data copy here\All design patterns\Design patterns and their code
s\Behavioral Design Pattern\8-State Pattern>javac StatePatternDemo.java

D:\all E drive data copy here\All design patterns\Design patterns and their code
s\Behavioral Design Pattern\8-State Pattern>java StatePatternDemo Sales
open database for sales
log activities
close the database
Sales has been updated

D:\all E drive data copy here\All design patterns\Design patterns and their code
s\Behavioral Design Pattern\8-State Pattern>java StatePatternDemo Management
open database for Management
log activities
close the database
Management has been updated

D:\all E drive data copy here\All design patterns\Design patterns and their code
s\Behavioral Design Pattern\8-State Pattern>java StatePatternDemo Accounting
open database for accounting
log activities
close the database
Accounting has been updated
```

ADVANTAGES:

- Organize the code related to particular states into separate classes.
- Introduce new states without changing existing state classes or the context.
- Simplify the code of the context by eliminating state machine conditions.