Design Patterns

State Pattern*

ebru@hacettepe.edu.tr
ebruakcapinarsezer@gmail.com
http://yunus.hacettepe.edu.tr/~ebru/
@ebru176

Kasım 2017

*modified from
http://*people.cs.uchicago.edu/~matei/TA/CSPP523*

General Description

- A type of Behavioral pattern.
- Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Uses Polymorphism to define different behaviors for different states of an object.

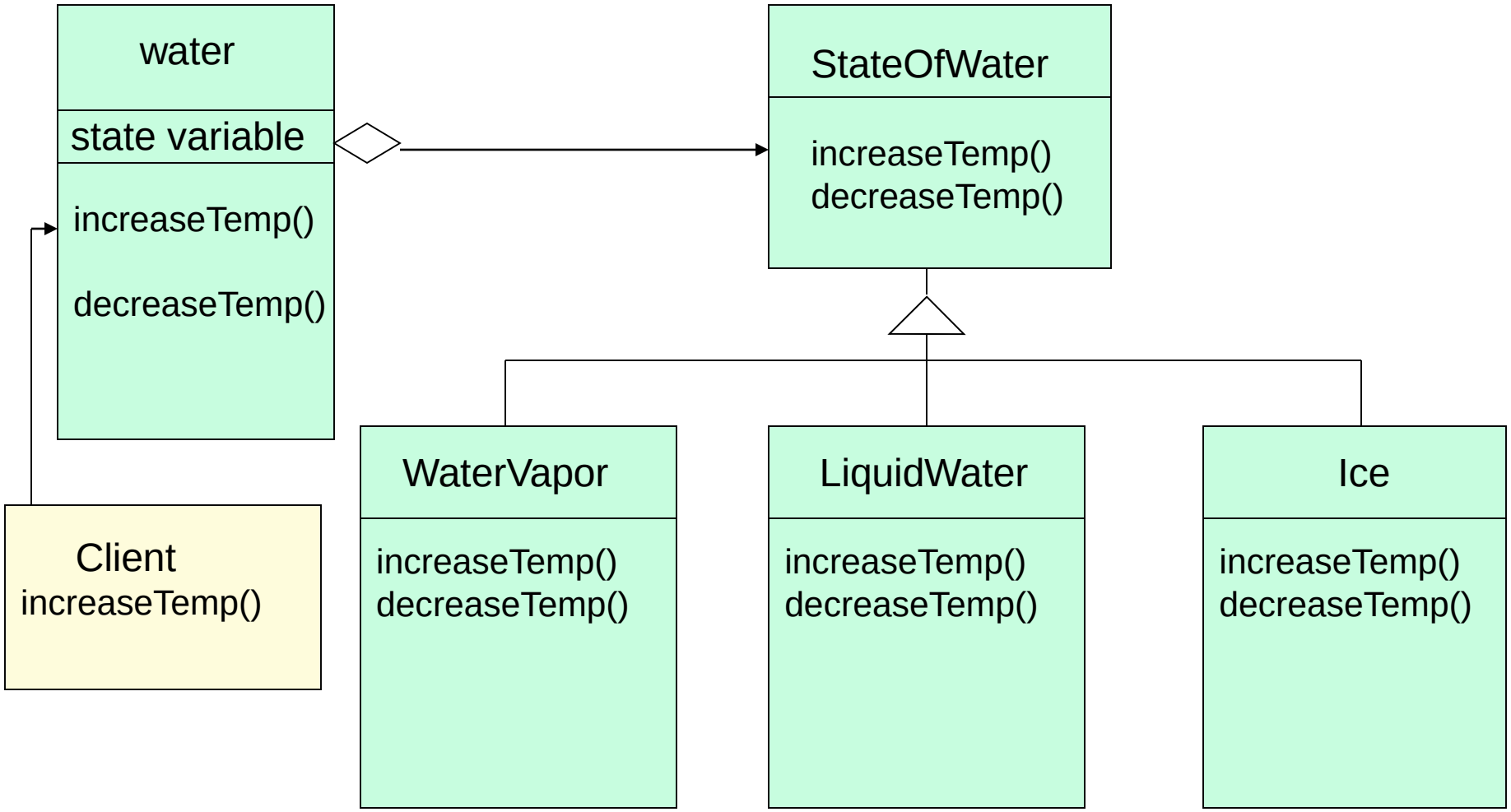
When to use STATE pattern ?

- State pattern is useful when there is an object that can be in one of several states, with different behavior in each state.

- To simplify operations that have large conditional statements that depend on the object's state.

```
if (myself = happy) then
{
   eatIceCream();
   ....
   }
else if (myself = sad) then
{
   goToPub();
   ....
   }
else if (myself = ecstatic) then
{
   ....
```

Example I

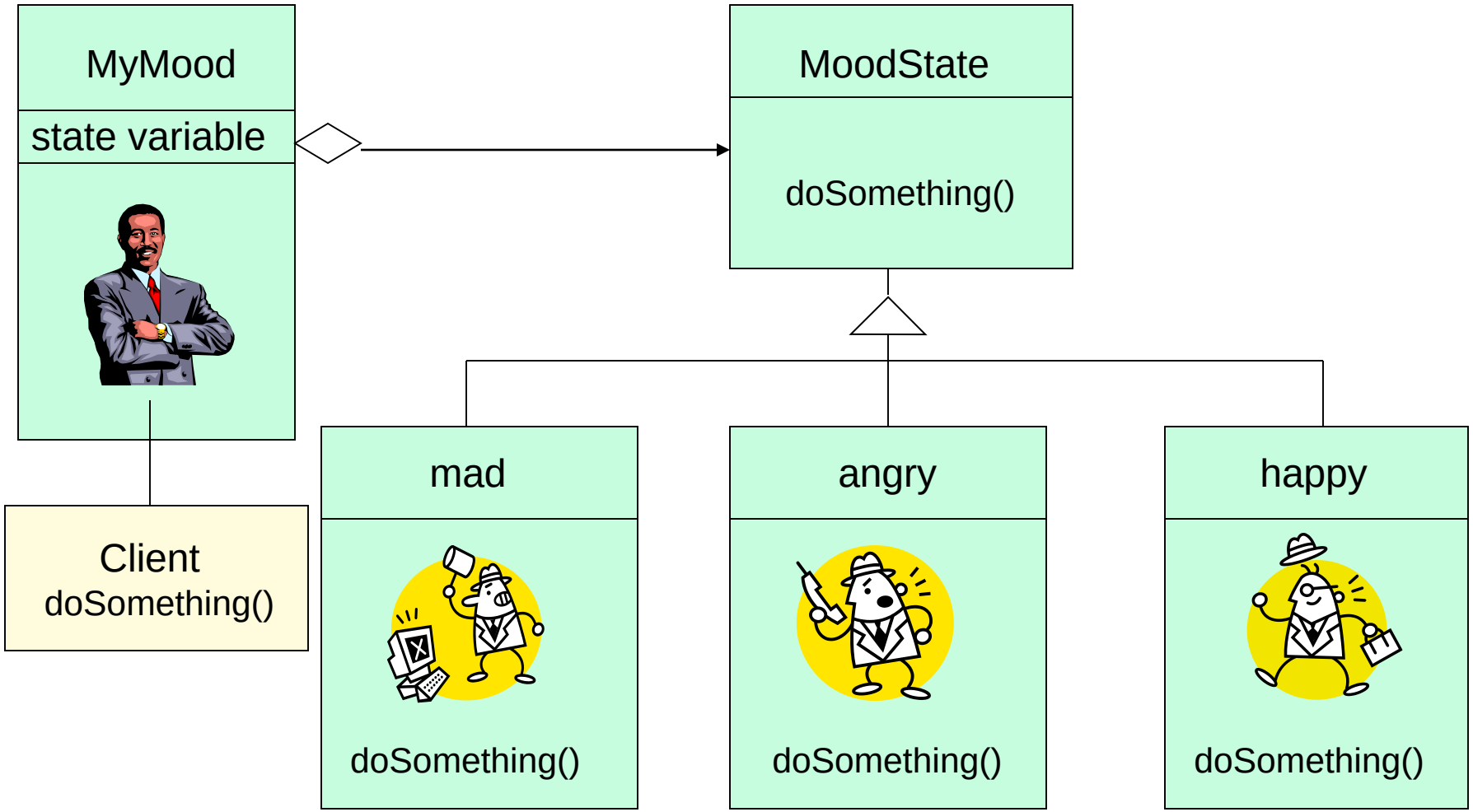

water
state variable
increaseTemp()
decreaseTemp()
StateOfWater
increaseTemp()
decreaseTemp()
Client
increaseTemp()
WaterVapor
increaseTemp()
decreaseTemp()
LiquidWater
increaseTemp()
decreaseTemp()
Ice
increaseTemp()
decreaseTemp()

How is STATE pattern implemented ?

- “Context” class:
  Represents the interface to the outside world.
- “State” abstract class:
  Base class which defines the different states of the “state machine”.
- “Derived” classes from the State class:
  Defines the true nature of the state that the state machine can be in.

Context class maintains a pointer to the current state. To change the state of the state machine, the pointer needs to be changed.

Example II

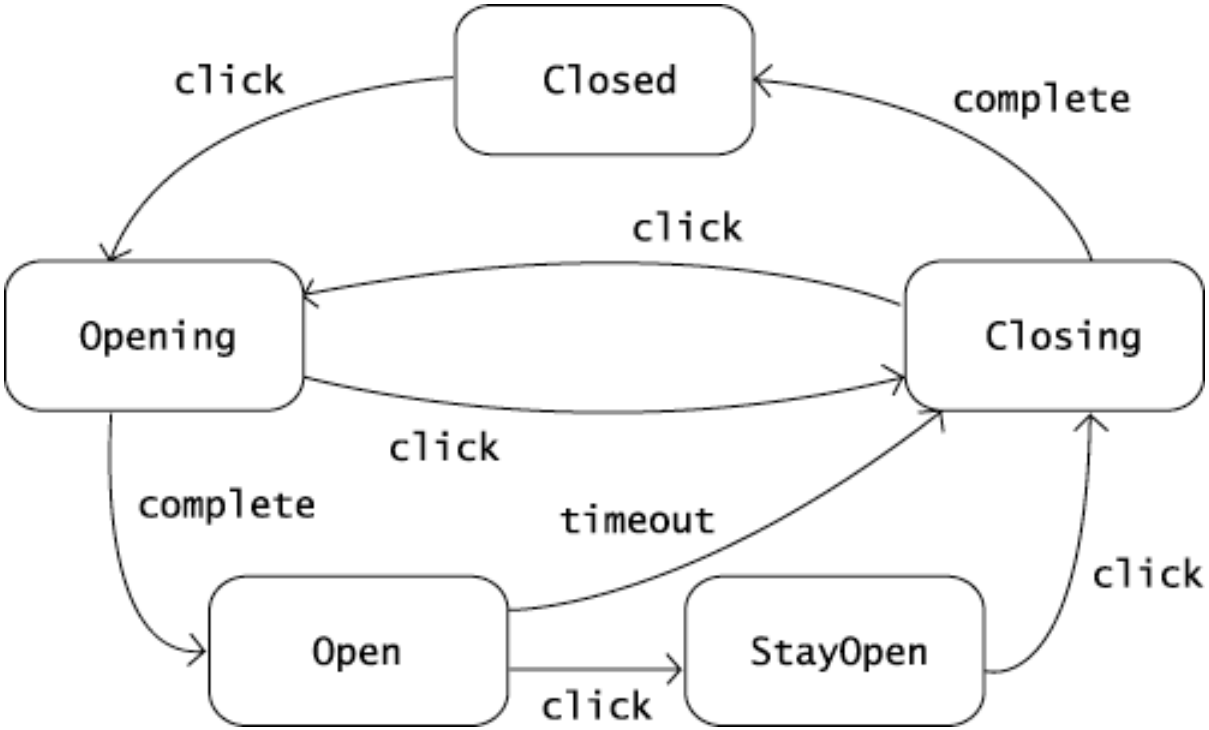

MyMood
state variable
MoodState
doSomething()
Client
doSomething()
mad
doSomething()
angry
doSomething()
happy
doSomething()

Possible Issues

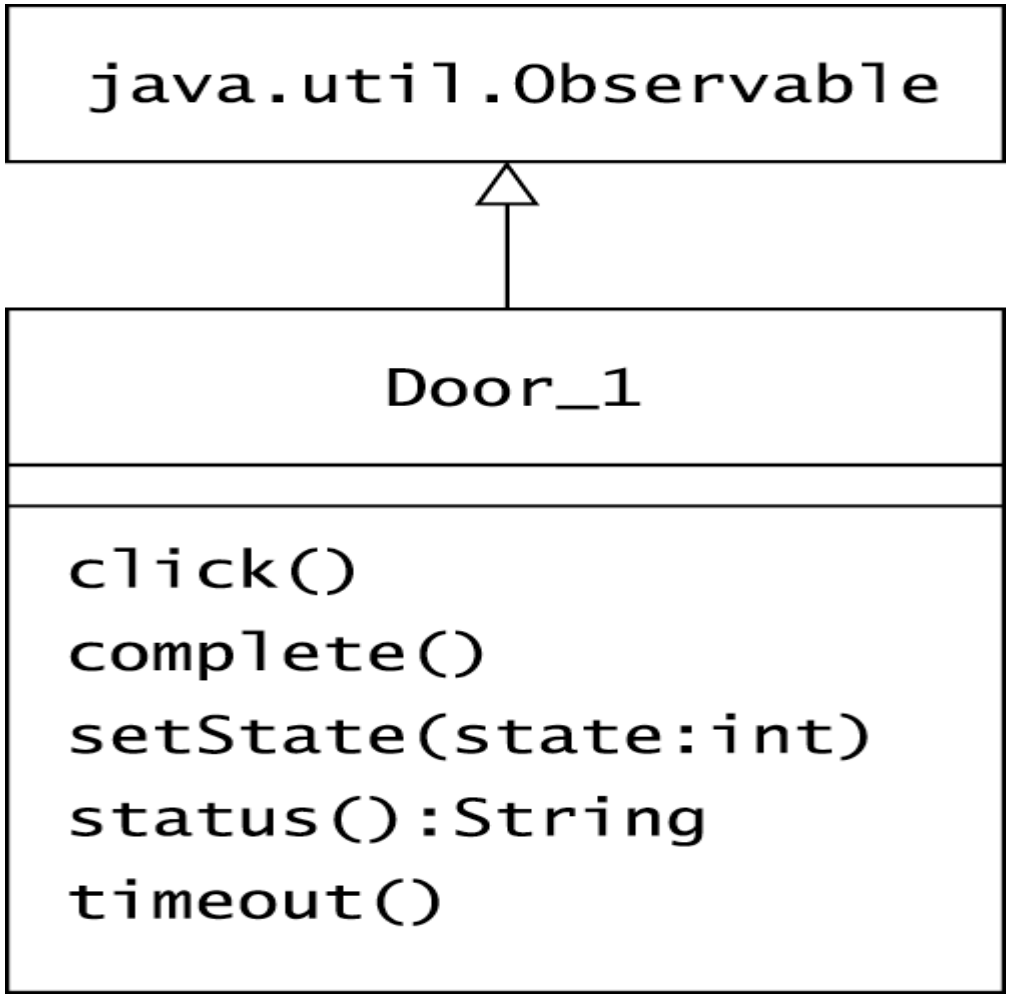
- If there are many different states that all need to have transition functions to one another, the amount of transition code required could be massive.

```
Switch(State)
  case A: …
  case B: …
  etc…
```

UML STATE DIAGRAM



Door_1 Class


java.util.Observable
Door_1
click()
complete()
setState(state:int)
status():String
timeout()

Door_1 Class

```
public class Door_1 extends Observable
{
    public static final int CLOSED = -1;
    public static final int OPENING = -2;
    public static final int OPEN = -3;
    public static final int CLOSING = -4;
    public static final int STAYOPEN = -5;
    private int state = CLOSED;

  public String status()
  {
    switch (state)
    {
      case OPENING : return "Opening";
      case OPEN : return "Open";
      case CLOSING : return "Closing";
      case STAYOPEN : return "StayOpen";
      default : return "Closed";
    }
  }
```

```
public void click()
  {
    if (state == CLOSED)
    { setState(OPENING); }
    else if (state == OPENING || state == STAYOPEN)
    { setState(CLOSING); }
   else if (state == OPEN)
   { setState(STAYOPEN); }
   else if (state == CLOSING)
  { setState(OPENING); }
 }

 private void setState(int state)
 {
   this.state = state;
   setChanged();
   notifyObservers();
 }
}
```

Refactoring to State Pattern

- Make each state of the door a seperate class

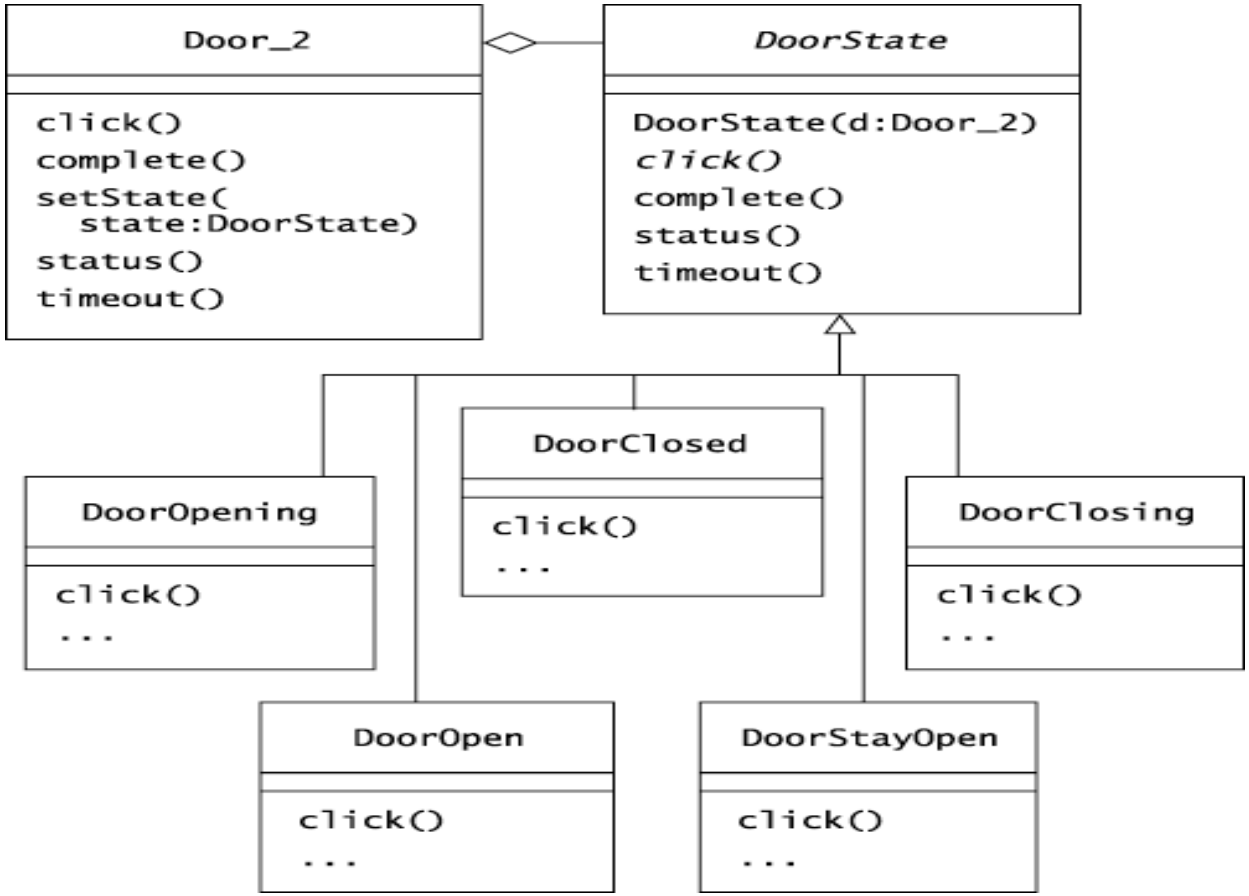


DIAGRAM SHOWS

- Door_2 class contains the context of the state
- DoorState class constructor requires a Door_2 object
- Subclasses of DoorState use this object to communicate changes in state back to the door

```
public class Door_2 extends Observable
{
    public final DoorState CLOSED = new DoorClosed(this);
    public final DoorState OPENING = new DoorOpening(this);
    public final DoorState OPEN = new DoorOpen(this);
    public final DoorState CLOSING = new DoorClosing(this);
    public final DoorState STAYOPEN = new DoorStayOpen(this);
    private DoorState state = CLOSED;

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

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

    protected void setState(DoorState state)
    {
       this.state = state;
       setChanged();
       notifyObservers();
    }

   public String status()
   {
      return state.status();
   }

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

}
```

```
public abstract class DoorState
{
      protected Door_2 door;

      public DoorState(Door_2 door)
      { this.door = door; }

      public abstract void click();
      public void complete() { }
      public String status()
      {
         String s = getClass().getName();
         return s.substring(s.lastIndexOf('.') + 1);
      }

     public void timeout() { }
}

public class DoorOpen extends DoorState
{
      public DoorOpen(Door_2 door)
     {
          super(door);
     }

     public void click()
     {
        door.setState(door.STAYOPEN);
     }

     public void timeout()
     {
         door.setState(door.CLOSING);
     }
}
```

Benefits of using STATE pattern

- **Localizes all behavior associated with a particular state into one object.**
  - New state and transitions can be added easily by defining new subclasses.
  - Simplifies maintenance.

- **It makes state transitions explicit.**
  - Separate objects for separate states makes transition explicit rather than using internal data values to define transitions in one combined object.

- **State objects can be shared.**
  - Context can share State objects if there are no instance variables.