13.5 THE STATE PATTERN

Several conventional state machine implementation approaches exist and some of them are widely used by programmers. However, the conventional approaches are associated with a number of drawbacks. The state pattern is a better alternative to the conventional approaches. This section reviews some of the conventional approaches and points out their drawbacks. It then presents the state pattern and shows its applications to the cruise control and the thermostat examples.

13.5.1 Conventional Approaches

Conventional approaches to implementing state behavior include:

1. Using nested-switch statements.

2. Using a state transition matrix.

3. Using method implementation.

The nested-switch implementation approach is widely used by programmers. In this approach, the current state is stored in a state variable. One switch statement is used to represent the states of the state machine. Another, nested, switch statement is used to represent the events. The cases of the inner switch are responsible for implementing the state transitions, that is, evaluating the guard conditions, invoking the response actions, and setting the state variable to the destination state. At runtime, the value of the state variable and the incoming event are used to select the appropriate case of the inner switch to execute.

The transition matrix approach stores the state transitions in the entries of a two-dimensional array, which is similar to the state transition table described in Section 13.4.3. Each array entry is either null or stores the guard condition, response actions, and destination state. The current state is kept in a state variable, which together with the incoming event, determines the array entry. If the array entry is not null, then the guard condition is evaluated; if it is true, the response actions are executed and the state variable is updated.

In the method implementation approach, the state behavior of a class is implemented by member functions that denote an event in the state diagram. The current state of the object is stored in a state variable, which is an attribute of the class. A member function may evaluate the state variable and the guard condition and execute the appropriate response actions and update the state variable.

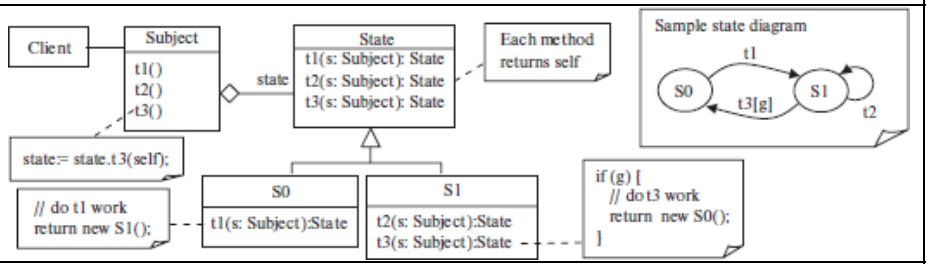
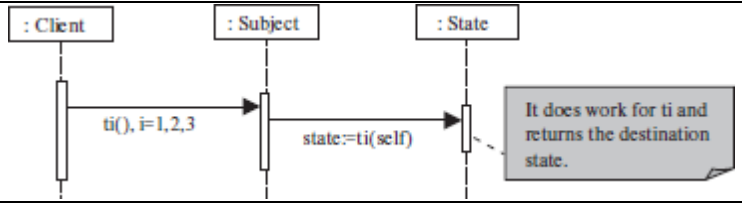
The conventional approaches are associated with a number of drawbacks. First, the nested-switch approach has high cyclomatic complexity, a measure of the number of independent control flow paths in a program. For example, if a state machine has five states and five events, then in the worst case, there are 5 by 5 or 25 independent paths. The method implementation approach has the same problem, although it is slightly better; the complexity is the number of states in the state machine. High

complexity makes the code difficult to comprehend, test, and maintain. The transition matrix uses an interpretation approach rather than a compilation approach. Therefore, it is not as efficient.

Second, changes are difficult to make. For example, using the nested-switch approach, each case of the outer switch must be changed in order to add or delete an event. If the changes are not made consistently, the system will not behave properly. Third, it is not easy to implement in the conventional approaches state diagrams that contain composite states. One problem is higher cyclomatic complexity introduced by composite states. Another problem is the difficulty in implementing concurrent state machines using the conventional approaches.

13.5.2 What Is State Pattern?

The state pattern, described in Figure 13.14, overcomes the drawbacks of the conventional approaches. The sample state machine shown in the structural design section is used to help understand the design of the pattern. The mapping between the sample state machine and the pattern classes is displayed in Figure 13.15. In practice, the concrete state objects may be created and stored in a static array in the State class. These can then be used to update the state of the subject to reduce the number of state objects created. Another approach uses the singleton pattern to implement the concrete states.

Name	State
Type	GoF/Behavioral
Specification	
Problem	How to design and implement state-dependent behavior of an object.
Solution	Define a State class that implements operations to correspond to the state transitions. Each of these operations simply returns self, that is, the current state. Define a State subclass for each state in the state diagram. Each State subclass overrides the parent-class operations corresponding to the outgoing transitions to implement the state behavior associated with these transitions.
Design	
Structural	
Behavioral	

Roles and Responsibilities	• State: It defines a method for each transition in the state diagram and implements it as "return self." • S0, S1: These are the State subclasses. There is one such subclass for each state in the state diagram. Each such subclass implements the transitions that go out of the state. The implementation does the work for each transition and returns the destination state. • Subject: It represents the object that possesses the state-dependent behavior. It provides the client interface and defines an operation for each of the transitions in the state diagram. It maintains a reference to a State object, which represents the state of the Subject. It delegates the client requests to the corresponding operations of the state object. The call automatically updates the state of the Subject. • Client: It represents the event sources such as window events, sensors, etc. It calls the operations of Subject that correspond to the events.
Benefits	• It reduces the complexity of the design and implementation of the state-dependent behavior. • It is easy to add and remove states and transitions. • It makes the state behavior easy to understand, implement, test and maintain. • It facilitates test-driven development because the behavior is easier to test.
Liabilities	More classes to design and implement.
Guidelines	• Apply the state pattern to objects that have nontrivial state behavior, or the state behavior will evolve in the future. • It is not necessary to apply the state pattern to state behavior that is trivial and not expected to evolve.
Related Patterns	• A use case controller may use state pattern to maintain use case states. • State subclasses are often implemented as singletons. • State and mediator patterns are often used together; the behavior of a mediator can be modeled by a state diagram, which is designed and implemented by using the state pattern.
Uses	The pattern is useful for design and implementation of event-driven systems, especially embedded systems.

FIGURE 13.14 Specification of the state pattern

Pattern Classes and Methods	Description	Sample State Diagram States and Transitions
State	An abstract state to represent all possible states.	
t1(): State t2(): State t3(): State	Methods of abstract State to correspond to the state transitions in the state diagram. Currently, there are only three transitions in the state diagram.	t1, t2, t3
S0, S1	Concrete states; for each state in the state diagram, there is a concrete State subclass in the pattern.	S0, S1
Subject	It represents the system, subsystem, device or object (e.g., a cruise control, a thermostat) whose state-dependent behavior is being modeled by the state diagram.	(implicitly assumed)
Client	It represents the environment of the Subject. It delivers the triggering events to the Subject by calling the corresponding functions of the Subject.	(implicitly assumed)

FIGURE 13.15 Mapping between state pattern and sample state machine

13.5.3 Applying State Pattern

As described in the previous section, the state pattern consists of two main parts: a subject with state behavior and a hierarchy of state classes that implements the state behavior for the subject. The subject defines the client interface and delegates client requests to the state object that keeps track of the state of the subject. Thus, the key to apply the state pattern is defining the hierarchy of state classes. This section illustrates the application of the state pattern to the cruise control and thermostat examples, respectively. **Figure 13.16** shows the state pattern for the Cruise Control in **Figure 13.11**. It includes UML notes to explain the various parts. Note in **Figure 13.14**, the state of the subject is updated by the return values of the functions of the State class. In **Figure 13.16**, the same effect is achieved in the body of the functions of the State class. For example, the UML notes attached to the Cruise Deactivated and the Increasing Speed classes show that the state of the Cruise Control is updated to Cruise Activated and Cruising, respectively. This example illustrates how to map a CSS and its substates to a state class and its subclasses in the state pattern. Note that a state machine itself is a CSS. The main points of the mapping are outlined as follows:

1. Define a Subject class with client interfaces to correspond to the state machine. The interfaces are identified from the events coming into the subject or one of its components in the domain model constructed in **Section 13.4.2**. Each of these client interface functions delegates the request to the State object. For example, the domain model in **Figure 13.6** shows eight events going into the Cruise Control software; therefore, the Cruise Control subject class in **Figure 13.16** has eight

functions. The functions can also be identified from the state diagram, that is, from the events that label the transitions in the state diagram.

2. Create a root class for the hierarchy of state classes to represent the states of the Subject. The member functions of this root class are identified from the client interfaces of the Subject class, the domain model, or the state machine as described above. Each of these member functions has a do-nothing implementation.

3. For each CSS of the state machine, create a state class with subclasses representing its substates and name the state class and the subclasses accordingly. If the CSS is a substate of another CSS, then make this newly created state class a subclass of the state class representing the containing CSS. For example, the substates of **Cruise Activated** are mapped to subclasses of the Cruise Activated class in the state pattern. The **Cruise Activated** and **Cruise Deactivated** states are mapped to two subclasses of the State class because the state machine itself is a CSS. For each substate S and each triggering event that labels a transition going out of S, override the corresponding member function in the state class corresponding to S.

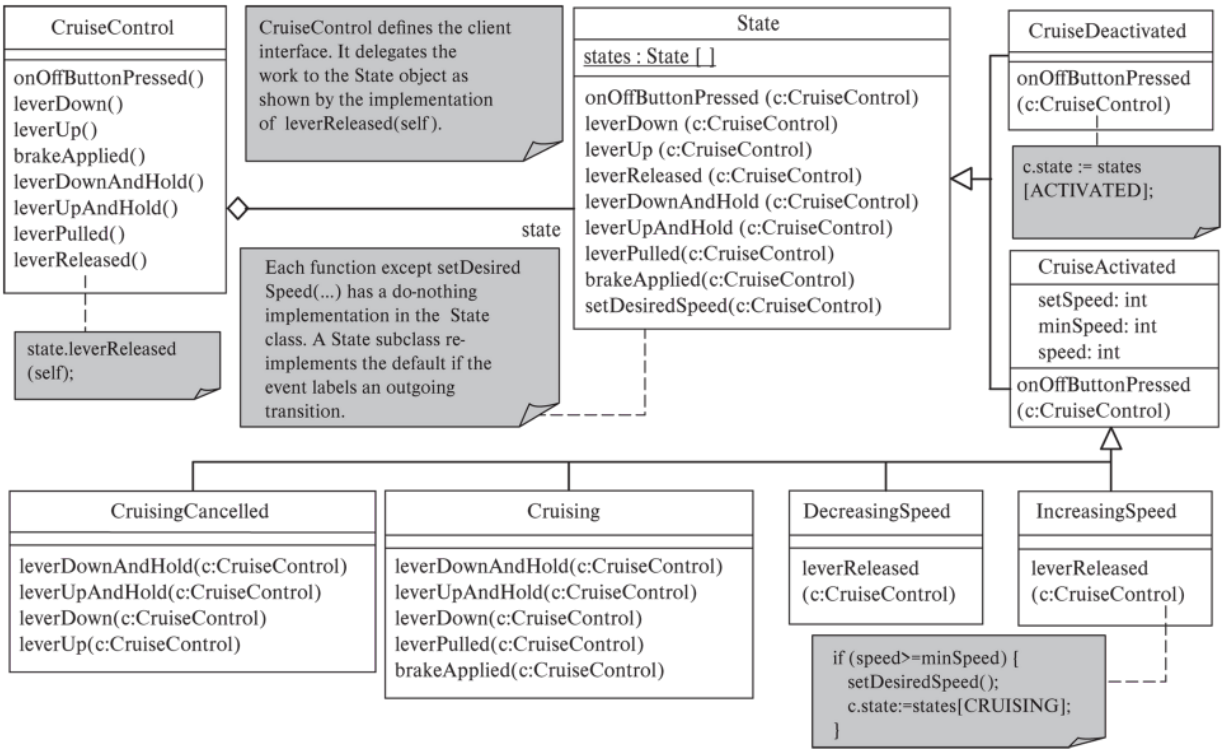


FIGURE 13.16 Cruise control state pattern

Figure 13.17 shows the state pattern for the Thermostat Control state diagram in Figure 13.13. This example shows that applying the state pattern to a state diagram containing a CCS is similar. The differences are as follows:

- The Subject class maintains N component state objects; each corresponds to a component substate of the CCS. Each client interface function delegates the request to the corresponding component state object.
- Each component has its own hierarchy of state classes as shown in Figure 13.17.

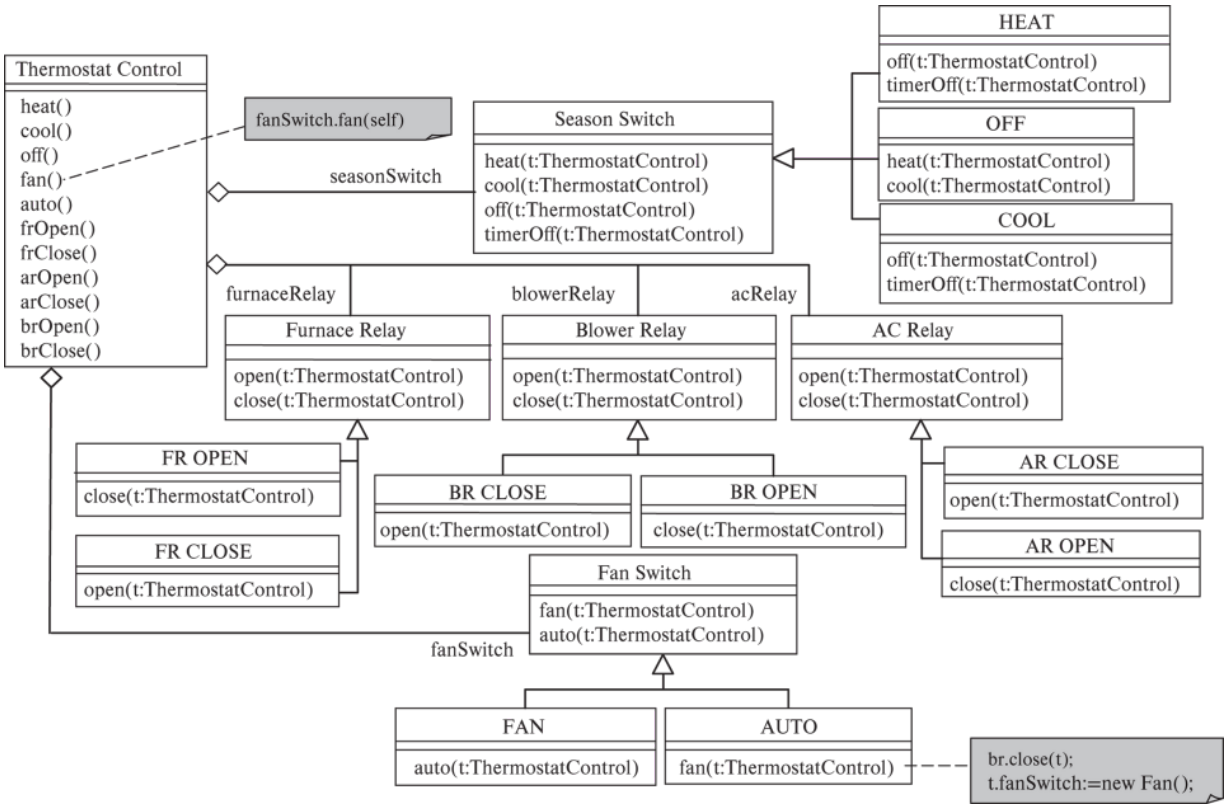


FIGURE 13.17 State pattern for the Thermostat Control

EXAMPLE 13.8

An automatic mower can mow a rectangular lawn one row after another starting from the upper-left corner. Figure 13.18 shows a state diagram describing the state behavior of the lawn mower. It uses a timer to generate a timer off event every second. When this happens, the mower checks if a border is ahead. If not, it cuts forwards, else it makes a right turn or left turn and moves down to cut the next row. Figure 13.19 shows the application of the state pattern to the state diagram in Figure 13.18 and Figure 13.20 shows the implementation of the state pattern in Figure 13.19. The Frame1 class provides a window that contains a start and stop buttons. The Canvas class is responsible for displaying the movements of the mower and the lawn

being cut. The Mower class holds its position, cuts the grass, and moves forwards, as well as checking the borders of the lawn.

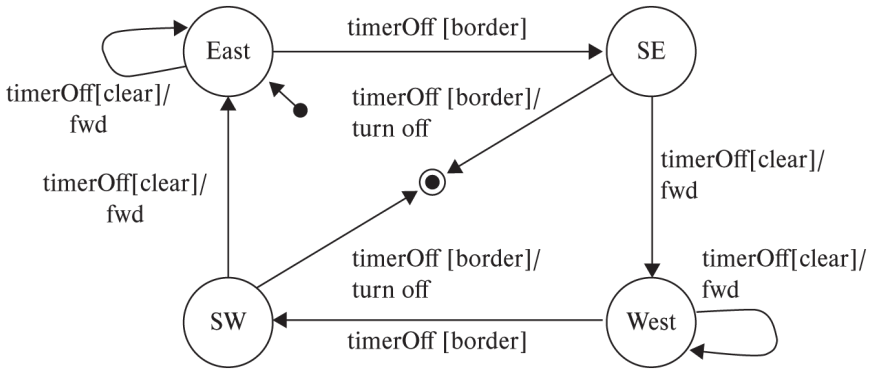


FIGURE 13.18 State behavior of the automatic mower

```
public class Mower extends Observable
  implements ActionListener {
  public static final int WIDTH=500, HEIGHT=300;
  private Timer timer=new Timer(1000, this);
  private int x=0, y=0;
  State state=new East();
  public void actionPerformed(ActionEvent ae) {
    state=state.timerOff(this);
  }
  public int getX() { return x; }
  public int getY() { return y; }
  public boolean clear(East e) { return x<WIDTH-50; }
  public boolean clear(SE e) { return y<HEIGHT-50; }
  public boolean clear(West e) { return x>0; }
  public boolean clear(SW e) { return y<HEIGHT-50; }
  public void fwd(East e) {
    x+=50; setChanged();  notifyObservers(); }
  public void fwd(West e) {
    x-=50; setChanged();  notifyObservers(); }
  public void fwd (SE e) {
    y+=50; setChanged();  notifyObservers(); }
  public void fwd(SW e) {
    y+=50; setChanged(); notifyObservers(); }
  public void stop() { timer.stop(); }
  public void start() { x=0; y=0; timer.start(); }

class State {
  public State timerOff(Mower m) {  return this; }
}

class East extends State {
   public State timerOff(Mower m) {
    if (m.clear(this)) {
      m.fwd(this); return this;
    } else  {  return new SE(); }
   }
}

class West extends State {
   public State timerOff(Mower m) {
    if (m.clear(this)) {
      m.fwd(this); return this;
    } else { return new SW(); }
   }
}

class SE extends State {
   public State timerOff(Mower m) {
    if (m.clear(this)) {
      m.fwd(this); return new West();
    } else { m.stop(); return this;  }
   }
}

class SW extends State {
   public State timerOff(Mower m) {
    if (m.clear(this)) {
      m.fwd(this); return new East();
    } else { m.stop(); return this; }
   }
}}
```

```
public class  MowerGUI extends JFrame {
  Mower am; Canvas canvas;  JButton start, stop;
  public MowerGUI() {
    am=new Mower();
    start=new JButton("Start");
    start.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent  e) {
        canvas.cuts.clear();  canvas.x=0; canvas.y=0;
        am.start(); canvas.repaint(); }});
    stop=new JButton("Stop");
    stop.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        am.stop(); }});
    getContentPane().setLayout(new BorderLayout());
    JPanel leftPane=new JPanel(new FlowLayout());
    getContentPane().add(leftPane,
      BorderLayout.NORTH);
    leftPane.add(start); leftPane.add(stop);
    canvas= new Canvas(); am.addObserver(canvas);
    canvas.setPreferredSize(new
      Dimension(Mower.WIDTH, Mower.HEIGHT));
    getContentPane().add(canvas,
      BorderLayout.CENTER);
    canvas.setVisible(true);
  }
  protected void processWindowEvent(WindowEvent e)
  {   super.processWindowEvent(e);
      if (e.getID()==WindowEvent.
        WINDOW_CLOSING) System.exit(0); }
  public static void main(String[] args) {
    MowerGUI frame=new MowerGUI();
    frame.setSize(new  Dimension(Mower.WIDTH,
      Mower.HEIGHT+100));
    frame.setLocation(200, 200);
    frame.pack(); frame.setVisible(true);  }

public class Canvas extends JPanel implements
Observer {
  int x, y;  ArrayList cuts=new ArrayList();
  public Canvas() {
    setBackground(new Color(0,100,0)); }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Cut s=new Cut(x, y); cuts.add(s);
    for(int i=0; i<cuts.size(); i++) {
      Cut x=(Cut)cuts.get(i); x.draw(g);
    }
  }
  public void update(Observable o, Object arg) {
    x=((Mower)o).getX(); y=((Mower)o).getY();
    repaint(); }
}

public class Cut {
  int x, y;
  public Cut(int x, int y) {
    this.x=x; this.y=y; }
  public void draw(Graphics g) {
    g.setColor(Color.GREEN);
    g.fill3DRect(x, y, 50, 50, false); }
}}
```

FIGURE 13.19 Applying state pattern for the automatic mower shown in Figure 13.18

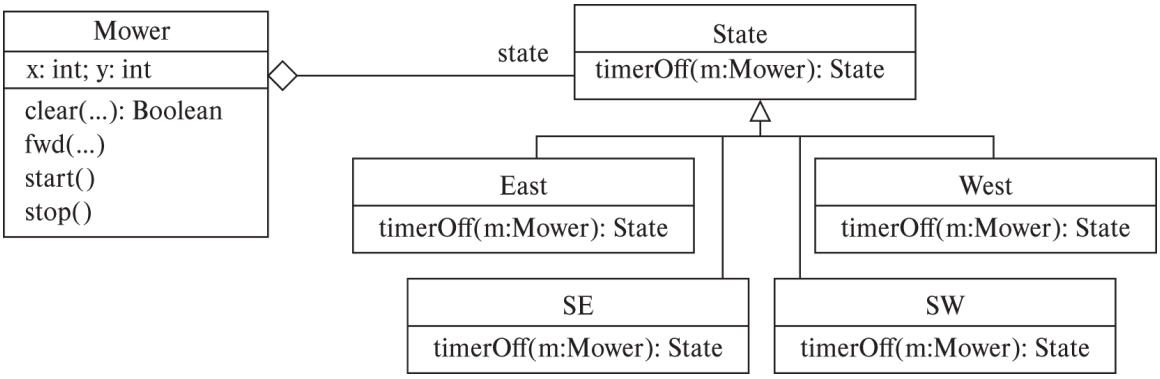


FIGURE 13.20 Implementation of the state pattern in Figure 13.19