An Integrated Approach to Object-Oriented Software Design

Brahma Dathan Sarnath Ramnath

April 15, 2020

Contents

Chapter 10

Modeling with Finite State Machines

10.1 Introduction

Our discussion thus far of the object-oriented software construction process has focused on the use case model. While this is a comprehensive technique that finds widespread application, it is inadequate for handling situations where the operations cannot be modeled by end-to-end use cases. This is typically the case with dynamic systems that respond to external input in real-time.

In this chapter, we present a case-study of a system where the use case model does not suffice: a controller for a microwave. (This analysis could be extended to most devices that interact with external entities.) The behavior of the microwave in response to a user's action depends on what state the microwave is in. For instance, if a cook/start button is pressed, the power tube that provides the heat does not always start operating. The case study starts by presenting a model for dealing with this kind of conditional behavior, and then goes on to discuss issues arising in the design and implementation of such systems.

Another commonly occurring situation is the creation of a Graphical User Interfaces(GUI). The program that implements the GUI typically presents different screens at different stages of the interaction. What screen gets displayed depends on the kind of input the application is requesting at that instant. While the underlying application itself may be designed using the use case model, the GUI is modeled as a system that changes its "appearance" in response to the interaction. It turns out that a similar model is useful for analyzing such systems. We will outline an approach to handle such programs in this chapter and explore the issue more fully in Chapter 9.

The term **embedded system** is used to refer to a system with a dedicated function, as opposed to a general purpose computer. An example of an embedded system is a CD player. It includes mechanical devices to rotate a CD, electrical devices to supply power, sense CDs, etc., and software to support the reading of the music in the CDs. The software that controls the hardware devices is termed **embedded software**. The system as a whole has to behave in a prescribed manner, turning components on or off depending on the input and the other environmental variables. In this chapter, we discuss the analysis, design, and implementation of a similar system.

10.2 A Simple Example

Problem Consider a simple microwave oven whose behavior is governed by the following rules:

- The microwave has a door, a light, a power tube, a button, a timer, and a display.
- When the oven is not in use and the door is closed, the light and the power tube are turned off and the display is blank.
- When the door is open, the light stays on.
- If the button is pushed when the door is closed and the oven is not operating, then the oven is activated for one minute. When the oven is activated, the light and the power tube are turned on.
- If the button is pushed when the oven is operating, one minute is added to the timer.
- The timer ticks every second. When the oven is operating, the display shows the number of seconds of cooking time remaining.
- If the door is opened when the oven is operating, the cooking is interrupted and ends (power tube is turned off). (The cook time remaining is now zero.)
- When the cooking time is completed, the power tube and light are turned off.
- Pushing the button when the door is open has no effect.

If we attempt to model this system with use cases, we run into some difficulties. Consider the following set of scenarios:

Scenario 1:

open door → place food in oven → close door → push button → wait for cooking to finish → open door → remove food → close door

Scenario 2:

open door → place food in oven → close door → push button → wait for cooking to finish → open door → remove food and stir → place food in oven → close door → push button → wait for cooking to finish → open door → remove food → close door

Scenario 3:

open door → place food in oven → close door → push button → wait for 30 seconds → open door → remove food and stir → place food in oven → close door → push button → wait for 45 seconds → open door → remove food → close door → stir food → open door → place food in oven → close door → push button → wait for cooking to finish → open door → remove food → close door

Clearly, there is no set of standard "business processes" that can characterize the manner in which an actor interacts with system. What we observe instead is that we are dealing with a continual sequence of events, and the manner in which these events are processed depends on the state in which the system is. The system may also change state in response to these events. What this suggests is that in order to model the system behavior accurately, we should take a different approach.

A **finite state machine (FSM)** is an abstract machine that can be in exactly one of a finite number of states at any given time and can change its state in response to some external inputs; the change from one state to another is called a **transition**. An FSM is defined by a list of its states, its initial state, and the conditions for each transition.

Formally, an FSM is defined by a set of states, a set of input symbols and a set of transitions. Each transition is defined by a 4-tuple (s_i, s_f, I, O), where s_i is the initial state, s_f is the final state, I is the input that triggers the transition, and O is the associated output, if any. Two different formulations for FSMs can be found in the literature on automata theory. These are the *Mealy machine* and the *Moore machine*. In a Mealy machine, the output depends on the event and the current state; a Moore machine is a simplification in which the output depends only on the state. The two are equivalent as far as their power is concerned, i.e., any system that can be defined in one kind of machine can also be defined in the other, but the number of states and the transitions vary.

10.3 Requirements Analysis via Finite State Modeling

In the previous chapters, we modeled our applications as a set of use cases to determine the user requirements. Here, we model the microwave as an FSM to derive the system specifications. For this, we first try to identify the states in the FSM that would model the behavior of the microwave oven.

We first create a table with three columns as shown below. As we read the problem statement, we populate each row by filling one or more columns. The columns are filled as follows.

We look at each item in the problem description and divide the different parts of each statement into one of three categories.

- If the part is related to the status of the microwave, we put it in the first column, which has the header *Situation*. A rule of thumb for deciding what to put here is to ask whether the thing under consideration is relatively long-lasting. But not all long-lasting things find a place in this column. This will become clearer as we fill the entries.
- If the part is related to an action, we put it in the second column, which has the header *Action*. These are things that take place instantaneously.
- If the part is a consequence of doing the thing under *Action* to the item under *Situation*, we put it in the third column, which has the header *Consequence*.

Not every statement has three parts.

Let us take some examples. Consider the statement

> When the oven is not in use and the door is closed, the light and the power tube are turned off and the display is blank.

The phrase "When the oven is not in use and the door is closed" is a description of the situation of the microwave. The rest of the statement, "the light and the power tube are turned off and the display is blank" is the consequence. There is no action here. By distributing these two into the first and third columns, we get the first row of the table.

A similar analysis applies to the second statement, "When the door is open, the light stays on." This gets us the second row of the table.

The third statement is

> If the button is pushed when the door is closed and the oven is not operating, then the oven is activated for one minute.

Here the situation is that "the door is closed and the oven is not operating." The action is "the button is pushed." The consequence is "the oven is activated for one minute." This results in the third row.

A similar analysis applies to most of the statements. In the statement,

> The timer ticks every second. When the oven is operating, the display shows the number of seconds of cooking time remaining.

we can see that "the timer ticks every second" is instantaneous and belongs to the second column. The situation is "the oven is operating." The consequence is "the display shows the number of seconds remaining in the cooking process."

There is a bit of complication in the statement

> When the cooking time is completed, the power tube and light are turned off.

Clearly, the phrase "the power tube and light are turned off" is a consequence. But the phrase "When the cooking time is completed" could be treated as a situation of the microwave or as an action that happens within the microwave. We resolve this by having two rows in one of which the phrase is treated as a situation and the other in which it is used as an action.

Situation	Action	Consequence
the oven is not in use and the door is closed		the light and power tube are turned off, and the display is blank.
the door is open		the light is on.
the door is closed and the oven is not operating	the button is pushed	the oven is activated for one minute.
the oven is operating		the light and the power tube are turned on
the oven is operating	the button is pushed	one minute is added to the timer
the oven is operating	timer ticks	the display shows the number of seconds of cooking time remaining.
the oven is operating	the door is opened	cooking is interrupted and the power tube is turned off
the cooking time is completed		the power tube and light are turned off.
	the cooking time is completed	the power tube and light are turned off.
the door is open	Pushing the button	has no effect.

To avoid verbosity and simplify matters a little bit, let us do two things.

- Loosely equate the power tube and the microwave and treat them as synonyms.

- Replace some of the verbose phrases such as "the oven is not in use and the door is closed" by more compact ones. In this case, we can use the term "Idle; Door Closed." The words "the oven is operating" and "power tube is turned on" are replaced by "Cooking."

The revised table is given below.

Situation	*Action*	*Consequence*
Idle; Door Closed		the light is off and oven is idle, and the display is blank.
the door is open		the light is on.
Idle; Door Closed	the button is pushed	Cooking for one minute.
Cooking		the light is on
Cooking	the button is pushed	one minute is added to the timer
Cooking	timer ticks	the display shows the number of seconds of cooking time remaining.
Cooking	the door is opened	cooking is interrupted and the oven is turned off
the cooking time is completed		the oven is idle and light is turned off.
	the cooking time is completed	cooking is completed and light is turned off.
the door is open	Pushing the button	has no effect.

The first column shows the following entries.

- Idle; Door Closed
- Door Open
- Cooking
- Cooking time is completed

These are possible states of the microwave. The list is not necessarily complete.
Entries in the second column are possible events. These are:

- The button is pushed
- Timer ticks
- Cooking time is completed
- Door is opened

Entries in the third column are possible states and outputs. After eliminating some of the terms that essentially mean the same thing, we get the following list.

- the light is off
- the oven is idle

- display is blank
- the light is on
- cooking for one minute
- one minute is added to the timer
- the display shows the number of seconds of cooking time remaining
- cooking is interrupted
- cooking is completed

In the list derived from the last column, we examine each entry and classify whether it is an output or is related to one of the states derived from the first column.

- The entries, "the light is off," "display is blank," "the light is on," and "the display shows the number of seconds of cooking time remaining" are indications for the user about what the microwave id doing.
- The entries "cooking for one minute" and "one minute is added to the timer" deserve some thought. We identified "Cooking" and "Cooking completed" as possible states. The two entries from the third column are related to cooking in that they incorporate the cooking time remaining. We should ask ourselves how we want to do this incorporation. Two possibilities exist.
  - Have the states "Cooking" and "Not Cooking" and represent the cooking time as just an associated value with the state "Cooking."
  - Have an enumerable, but infinite number of states for each cooking time.

  Obviously, we don't want to have an infinite number of states. So we should have two states "Cooking" and "Not Cooking." But this suggests another issue. One might think that "Not Cooking" is essentially "Cooking" with 0 for the number of seconds remaining in the cooking process. But that is essentially "Cooking Completed" or interrupted. Thus we arrive at the following possibilities for states related to cooking.
  - Cooking
  - Cooking Completed
  - Cooking Interrupted

With the above refinements, our table becomes the following. (We have rearranged the table, so rows with all columns filled are at the top.)

We now examine the entries in the Situation column. The meaning of the states Idle; Door Closed and Cooking are obvious. Let us rename the state the cooking time is completed as Completed. For uniformity, we rename the state door open as Idle; Door Open.

Let us proceed to the entries in the second column. These are events. Instead of the longer term "the button is pushed," we will use the term "Cook." We shorten "the door is opened" by replacing it with "Open Door," which brings us to another point. Corresponding to the event "Open Door," there has to be an event "Close Door." We shorten the entry "the

Situation	*Action*	*Consequence*
Idle; Door Closed	the button is pushed	Cooking
Cooking	the button is pushed	Cooking
Cooking	timer ticks	the display shows the number of seconds of cooking time remaining.
Cooking	the door is opened	Cooking Interrupted
the door is open	Pushing the button	has no effect.
Idle; Door Closed		the light is off and oven is idle, and the display is blank.
the door is open		the light is on.
Cooking		the light is on
the cooking time is completed		the oven is idle and light is turned off.
	the cooking time is completed	cooking is completed and light is turned off.

Table 10.1: Final Table After Successive Refinements. This shows some of the state transitions and expected outputs in the states.

cooking time is completed " as "Timer Runs Out," which also helps us distinguish it from "Completed."

Finally, we move onto the third column. We see the Cooking state identified from the first column appearing here. The entries "the display shows the number of seconds of cooking time remaining," "the light is off", "the display is blank," and "the light is on" are clearly outputs. The entry "the oven is idle" corresponds to the state "Idle; Door Closed." Let us look at "Cooking Interrupted." This happens when we open the door while cooking is in progress. Cooking ends because we interrupted the process. We have already identified it as a state and we use the shorter term "Interrupted" from now on.

The states identified are:

1. Idle; Door Closed Microwave is idle and the door is closed.
2. Idle; Door Open Microwave is idle and the door is open.
3. Cooking
4. Interrupted Cooking is interrupted by the door being opened.
5. Completed Microwave has completed cooking.

These states are found by looking at the process of cooking, and viewing each step of the cooking process as a separate state. We have the following events that cause the microwave to change state:

1. Open Door door is opened
2. Close Door door is closed
3. Cook button is pushed

4. Timer Ticks
5. Timer Runs Out

The first three are external events, the result of the actions of an external user. The last two are internal events triggered by the operation of the microwave.

We can now construct the table in Figure 10.1, which describes all the actions that correspond to each (*state, event*) pair. The rows in the first column list the possible states of the microwave, while the columns in the first row show the possible events. The other non-blank cells show the state transitions. The table is called a **state transition table**. An example will make clear how to use this table. With the microwave in the Cooking state (see row 4, column 1) if the door is opened (row 1, column 2), the cell formed by row 4 and column 2 shows that the microwave enters the Interrupted state.

The information in Figure 10.1 can also be represented pictorially using what is called the **UML state transition diagram** as shown in Figure 10.2. Each rectangle with rounded corners corresponds to a microwave state. The directed arcs tell what the new microwave state will be when a certain event occurs in a given state. For instance, if the microwave is in the Idle; DoorClosed state (the rectangle at the top-left part of the diagram), one of the arcs leading from the rectangle shows that if the cook button is pressed, the microwave enters the Cooking state.

From Table 10.1, we can extract a table that omits all the state transitions.

State	*Output*
Idle; Door Open	the light is on.
Cooking	the light is on; the display shows the number of seconds of cooking time remaining
Idle; Door Closed	the light is off and the display is blank.
Completed	the light is off.

The above can be used in the implementation stage to ensure the output is correct.

10.3.1 State Minimization

One observation we make here is that the behavior of our finite state machine is identical in the states Idle; Door Closed and Completed. This tells that we do not need separate states to distinguish the behavior of the system when it is idle with door closed from the behavior when it has completed cooking. Likewise, states Idle; Door Open and Interrupted are indistinguishable; we can therefore combine these two states into a single state Door Open, and merge states Idle; Door Closed and Completed into a single state, Door Closed. In effect, we have simply dropped states Interrupted and Completed from our model. The reduced FSM is described by the table in Figure 10.3.

In general, it is important to find an FSM with a small number of states and there are exact algorithms for state minimization[1]. Usually, fewer states imply a simpler system that is easier to maintain, but in some situations, it may be helpful to add a few redundant states to improve the readability of the design as a whole.

[1]Many text books on *Digital Logic Design* or *Automata Theory* would have descriptions of state minimization algorithms.

	Open Door	Close Door	Cook	Timer Ticks	Timer Runs Out
Idle; Door Closed	Idle; Door Open	Idle; Door Closed	Cooking	Idle; Door Closed	Idle; Door Closed
Idle; Door Open	Idle; Door Open	Idle; Door Closed	Idle; Door Open	Idle; Door Open	Idle; Door Open
Cooking	Interrupted	Cooking	Cooking	Cooking	Completed
Interrupted	Interrupted	Idle; Door Closed	Interrupted	Interrupted	Interrupted
Completed	Idle; Door Open	Completed	Cooking	Completed	Completed

Figure 10.1: Transition table for the microwave. The cells in the first column (other than the blank cell in the top row) list the states. The column headers name the events. The remaining cells represent state transitions in response to events. For example, if the current state is Idle; Door Closed (the left-most column in the second row) and the event is Cook (the fourth column), the state changes to Cooking. Also note that in some states, for certain events, the next state is the same as the current state.

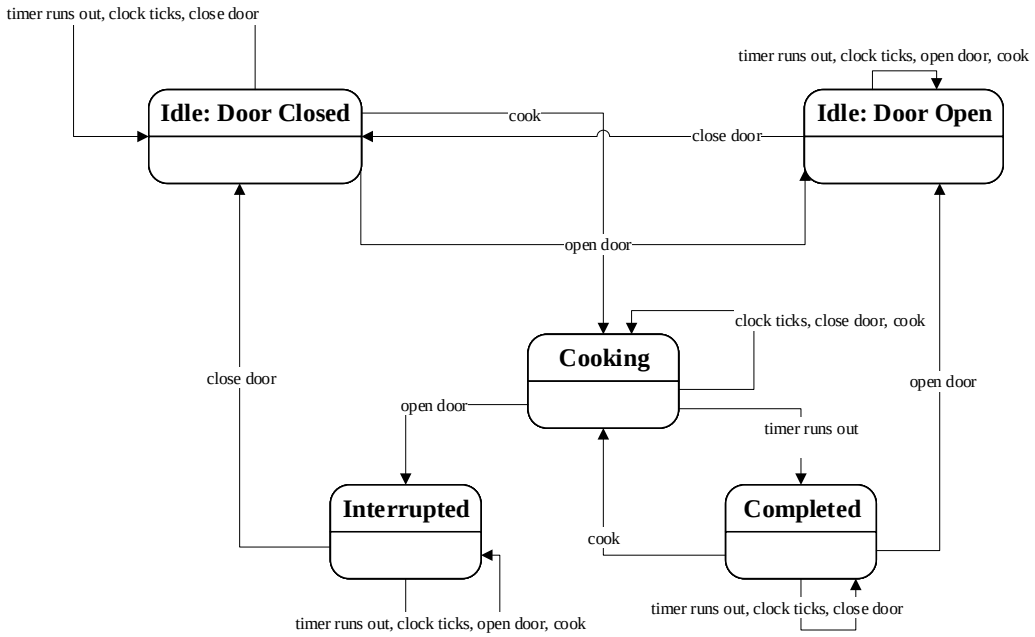

Figure 10.2: State Transition Diagram for the Microwave. Each rounded-rectangle represents a state. The name of the state is written in the rectangle. The arrows represent state transitions in response to events and the direction of the transition is indicated by the arcs. For example, if the current state is Idle; Door Closed and the event is Cook, the state changes to Cooking. Also note that in some states, for certain events, the next state is the same as the current state.

	Open Door	Close Door	Cook	Clock Ticks	Timer Runs Out
Door Closed	Door Open	Door Closed	Cooking	Door Closed	Door Closed
Door Open	Door Open	Door Closed	Door Open	Door Open	Door Open
Cooking	Door Open	Cooking	Cooking	Cooking	Door Closed

Figure 10.3: Minimized transition table for the microwave. We have collapsed the original table by combining two pairs of states: Idle; Door Open is equivalent to Interrupted, so they form the state Door Open; Idle; Door Closed is equivalent to Completed and they form the state Door Closed.

Use-case modeling *vs* **Finite State Modeling** One question that we need to address is *Under what conditions should we use FSMs, and under what conditions do we employ use cases?*

To help answer this question, let us examine how the library system designed in the previous chapters changes with each transaction. At the start of each use case (i.e., transaction), some pre-conditions hold. The final output of the transaction depends on the pre-conditions that were true at the start of the transaction. The state of the system defines (and is defined by) which pre-conditions are true. These pre-conditions, in turn, are determined by the values held by all of the objects in the system. Whenever a transaction is completed, as when a book is issued, the state changes because several objects, including the `Book` and the `Member` objects, get updated. (Note that this notion of state is somewhat different from the states of the FSM.) Each transaction has one "most-common" outcome (which we call the *main flow*) and other secondary outcomes, and the set of pre-conditions that hold decides the outcome of the transaction. If one were to model such a system by listing all the states and how we can switch between them via transactions, we would have a very complex structure with an unmanageable and possibly unbounded set of states because one could imagine books and members being added and deleted and updated throughout the life of the library system. On the other hand, the set of interactions that an actor can have with the library system is bounded. This indicates that we should prefer the simple functional specification provided by a use case model.

Contrast this situation with the microwave example. Here we have a possibly unbounded number of ways in which the actor can interact with the system. However, from the specifications, it is clear that we are only interested in how the system reacts to a given input (sometimes referred to as "reactive" systems). The nature of the reaction depends on the state the system is in (the word "state" being used to describe a behavioral response). Also we typically have a relatively small set of states in which the system could be at any point, and a clear set of transitions between them, which are triggered by events. This indicates that use case modeling is inappropriate and that using an FSM to model the system behavior would be the best choice.

Constructing State Transition Diagrams (Tables) A beginner often finds it somewhat difficult to construct state transition diagrams (or tables). An approach that helps is given in Section 10.3. The discussion in this box should gives some more insight into developing these diagrams. As you gain experience, you will find the process a lot easier. To develop state transition diagrams, we have to identify the states and events and determine the transitions. As you follow the procedure mentioned above, ensure that the states you identify have the following characteristics.

- *States are mutually exclusive.* If S_1 and S_2 are two states, it should be impossible for the system to be in both of them at the same time. For example, it is impossible for the microwave to be in the states IDLE; DOOR OPEN and IDLE; DOOR CLOSED at the same time. In contrast, we cannot have two states COOKING and DOOR CLOSED (assuming the obvious semantics) because the microwave could have the door closed and still be cooking, which violates the mutual exclusion property. But just because two things are mutually exclusive, there is no implication that they qualify to be states. For example, again assuming the obvious meanings DOOR CLOSED and DOOR OPEN are mutually exclusive, but we don't think of DOOR CLOSED as a state.

- *If two situations result in different behaviors, they possibly mean different states.* For example, IDLE; DOOR CLOSED and IDLE; DOOR OPEN are different states because in the former the light is off and in the latter, the light is on. In the former, the microwave responds to the pushing of the cook button.

- *A system remains in a state unless an event happens and there is no ambiguity as to which state the transition would be.* We are talking of deterministic FSMs, where a given state and event uniquely determines the next state. For example, the light being on remains that way until an event happens, but it is not a state because it is unclear what action will turn the light off as the light can be on when the microwave is cooking as well as when the door is open. The light on phenomenon is an output of the microwave, not a state.

The basic characteristics of an event are:

- *It affects the FSM in some way in the context of the problem.* For example, pressing the Cook button would have an effect, unless the door is open.

- *It is more or less instantaneous in nature.* For example, closing the door falls in this category. Not all things that are instantaneous are events. For example, a display that says there are 12 seconds remaining in the cooking process occurs for a very short time, but is an output of the microwave and violates the previous characteristic.

Once you get a correct set of states and events and fully understand what they mean, the state transition diagram is relatively easy to construct.

10.4 A First Solution to the Microwave Problem

We now proceed to complete the analysis of the microwave system to come up with a set of conceptual classes. As we did in the case of the library system, we then identify and design the physical classes and implement the system.

10.4.1 Completing the Analysis

Having created a model, the next step in our analysis is to identify the conceptual classes. As we did in Chapters 6 and 7, we start by constructing the list of nouns: microwave, power tube, light, display, door, cook button, etc. The display and cook button will be part of the user interface (GUI). Since this is only a "software simulation," we cannot have a real power tube or light, and so we simply have to model these by displaying some message on the GUI. (To make it more realistic, one could imagine that a system has software drivers to manipulate these devices, and these drivers are being invoked from the controller.) Likewise, the opening and closing of the door is simulated by some GUI component(s). This leaves us with a class for the microwave, and one for the GUI. The noun "timer" suggests that we need some mechanism to monitor the passage of time. The microwave can keep track of the time remaining with a field, but will have to be informed about the events that mark each unit of time. This can be done by a clock that generates ticks at regular (viz., one second) intervals. The conceptual classes are:

1. **Microwave.** This has the responsibility of keeping track of what state the oven is in, and for turning the power tube and light on/off. The oven must listen to the following events: *Opening/closing of the door, pushing of the button, and the timer running out.*
2. **GUI Display.** As described above, the GUI has components for user input, and will display some information to simulate operation. This suggests the following four displays:
   (a) One of the displays tells us whether the unit is cooking or not cooking.
   (b) A second display informs us whether the door is open or closed.
   (c) The third display shows the time remaining for cooking. If the microwave is idle, the display shows 0.
   (d) The fourth display gives the status of the light: whether it is on or off.
3. **Clock**. This is a class that generates clock tick events at regular intervals.

10.4.2 Designing the System

The first step in the design is to identify the software classes. This is an easy task here since the conceptual classes could very well be the physical classes. The next step is to figure out how the software classes will distribute the responsibilities to achieve the behavior specified in the model. In our case, this amounts to specifying how the events will be processed. We have two kinds of events:

- *User inputs:* These are recognized by the GUI.

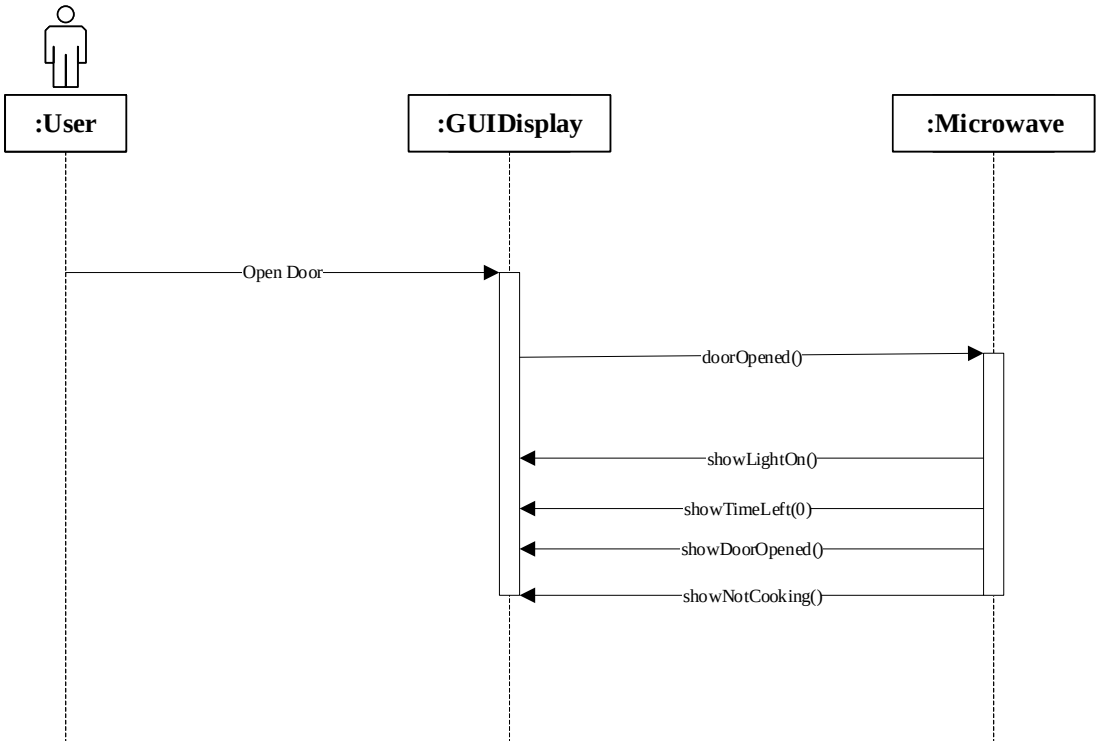

Figure 10.4: Sequence Diagram for the Door Open event. When the user clicks the Door Open button, the `GUIDisplay` object calls the `doorOpened()` method of the `Microwave` class., which issues a sequence of calls to the `GUIDisplay` object.

- *Clock ticks:* These originate in `Clock.`

To describe the manner in which the entities of the system handle these, we use sequence diagrams. Figure 10.4 shows how the system handles the user input corresponding to the opening of the door. The diagram suggests that we have a separate method in the `Microwave` class for each kind of event that occurs in the GUI. The actor can see the result of the action through the updates occurring on the display.

The sequence diagram for the other inputs from the user look similar. In each case, `Microwave` does some processing and updates the display. There are several methods to update the different aspects of the display and the appropriate ones will be invoked with the necessary parameters.

Figure 10.5 describes how the system handles a clock tick. Note that unlike the sequence diagrams we have seen earlier, this interaction is not initiated by the actor.

The sequence diagrams for the other events are similar, and they give us enough information to specify the responsibilities of the individual classes. `Microwave` is a singleton class with methods to process the events (door opening, clock tick, etc.) To mimic the FSM, we keep a variable `currentState` that keeps track of the state the microwave is in. We also need a variable that keeps track of the time remaining for cooking. The class diagram is shown in

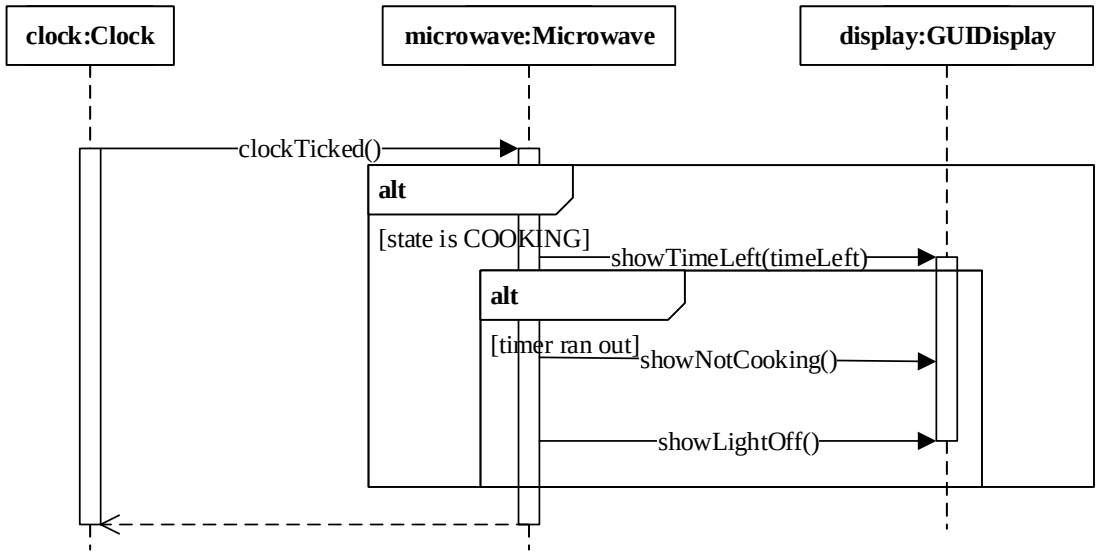

Figure 10.5: Sequence Diagram for processing a clock tick. The event is generated within the clock. The event is sent to the `Microwave` object, which ignores it unless the current state is the cooking state. If it is the cooking state, the remaining cooking time is updated by calling the `showTimeLeft()` in the `GUIDisplay` object. If the time remaining is 0, the light is turned off and the status is shown as not cooking.

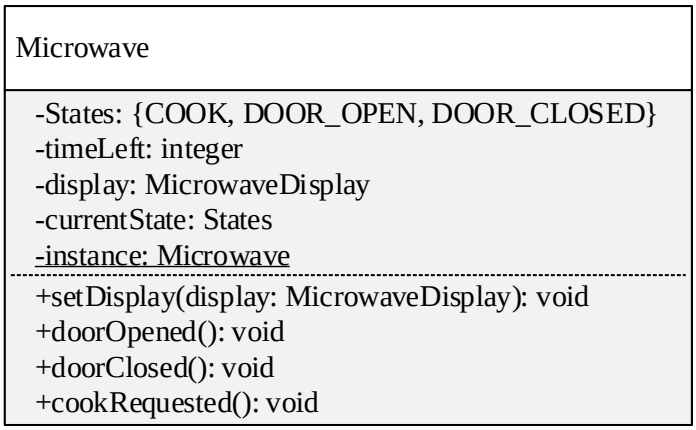

Figure 10.6: Microwave Class Diagram. The class is a singleton (see the static field and method) and contains one instance method for each event: when the clock ticks, however, the timer runs off event is "internally" generated.

Figure 10.6.

Although a text-based interface is difficult, if not impossible (no buttons), any number of graphical interfaces are possible. As shown in the sequence diagram for opening the door, the display class must provide methods of two kinds:

- Methods that process the input provided by the user.
- Methods that can be invoked by `Microwave` to display output. The reader is invited to draw the sequence diagrams for door closing and the push of the cook button and compile the set of methods in this category.

Methods of the first kind are largely defined by the kind of look and feel desired for the user interface. Methods of the second kind represent the functionality required by `Microwave`. When the door is opened, for instance, `Microwave` requests the display to indicate that the light is on. One way we could do this is to have `Microwave` get a reference to the appropriate object within the GUI and set it; such an approach results in `Microwave` being tied to one kind of look and feel and any attempt to change the look and feel will require changes to `Microwave`. To avoid such tight coupling between the GUI and `Microwave`, the essential functionality is abstracted out in the interface `MicrowaveDisplay` shown in Figure 10.7.

The class `GUIDisplay` implements this interface.

The `Clock` class has to initiate an event at regular intervals, so we model it as a thread. As shown in Figure 10.8, it implements the `Runnable` interface.

10.4.3 The Implementation Classes

We are now ready to work out the implementation details. The `Clock` class is the simplest and is therefore a good place to start. The `run` method is an infinite loop waking up every

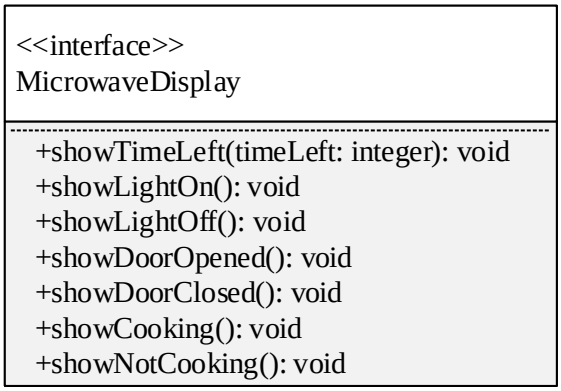

Figure 10.7: Microwave display is an interface that specifies the functionality of any user interface for the microwave application. We expect the user interface to be a simple GUI or one that resembles a real microwave or anything in between.

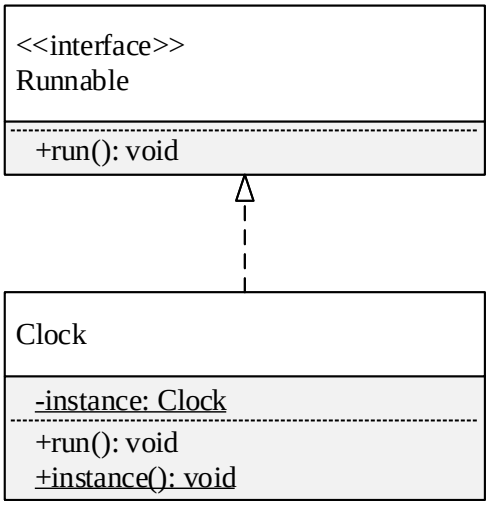

Figure 10.8: The Clock class is a thread that is an infinite loop, sending notifications to the Microwave object every second. It is a singleton, which accounts for the static members.

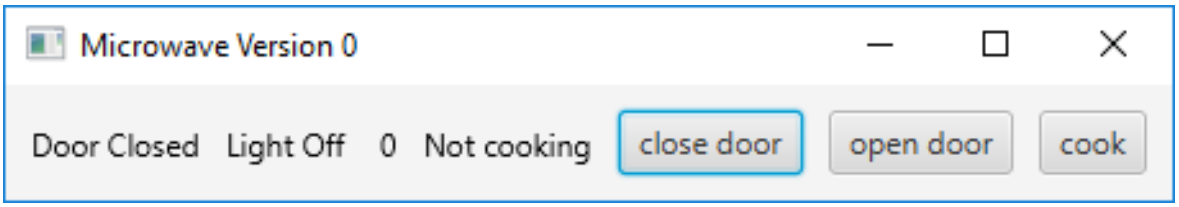

Figure 10.9: Microwave Interface: Note that the door is "simulated" by two buttons: one to open and the other to close the door.

second and invokes the `clockTicked` method on the `Microwave` object. The code is given below.

```
public class Clock implements Runnable {
    private static Clock instance;
    private Clock() {
        new Thread(this).start();
    }
    public static Clock instance() {
        if (instance == null) {
            instance = new Clock();
        }
        return instance;
    }
    public void run() {
        try {
            while (true) {
                Thread.sleep(1000);
                Microwave.instance().clockTicked();
            }
        } catch (InterruptedException ie) {
        }
    }
}
```

The Display Class

`GUIDisplay` is the concrete class that implements `MicrowaveDisplay`. To handle user input, it creates a `Stage` with a `Button` for each kind of operation: open door, close door, and cook. When run, the program displays the interface given in Figure 10.9. It has `Text` fields for displaying the status.

```
public class GUIDisplay extends Application implements MicrowaveDisplay,
                                                EventHandler<ActionEvent> {
    private Button doorCloser;
    private Button doorOpener;
    private Button cookButton;
```

```
    private Text doorStatus = new Text("Door Closed");
    private Text timerValue = new Text("            ");
    private Text lightStatus = new Text("Light Off");
    private Text cookingStatus = new Text("Not cooking");
    // other fields and methods
}
```

The `start` method lays out all the widgets and sets the `GUIDisplay` object to be the `EventHandler` for all the `Button` object clicks.

```
public void start(Stage primaryStage) throws Exception {
    display = this;
    doorCloser = new Button("close door");
    doorOpener = new Button("open door");
    cookButton = new Button("cook");

    GridPane pane = new GridPane();
    // code to set gaps between elements not shown
    pane.add(doorStatus, 0, 0);
    // code to add other widgets
    showDoorClosed();
    showLightOff();
    showTimeLeft(0);
    doorCloser.setOnAction(this);
    doorOpener.setOnAction(this);
    cookButton.setOnAction(this);
    Scene scene = new Scene(pane);
    primaryStage.setScene(scene);
    primaryStage.setTitle("Microwave Version 0");
    microwave = Microwave.instance();
    microwave.setDisplay(this);
    primaryStage.show();
    primaryStage.addEventHandler(WindowEvent.WINDOW_CLOSE_REQUEST, new EventHandler<WindowEvent
       public void handle(WindowEvent window) {
           System.exit(0);
      }
    });
}
```

Turning our attention to event processing, we note that the general strategy in each case is to check the current state and take the appropriate action. If the action results in a change of state, some transitional work also needs to be done. As an example, consider the `cookRequested()` method. The cooking request is processed only if the system is cooking (`COOKING_STATE`) or if the door is closed (`DOOR_CLOSED_STATE`). In case of the latter, `currentState` is first changed to `COOKING_STATE` and the necessary transitional operations are performed. In case of the former, 60 seconds are added to the time remaining and the display is updated.

```
   public void cookRequested() {
```

```
        if (currentState == States.DOOR_CLOSED_STATE) {
            currentState = States.COOKING_STATE;
            display.showCooking();
            display.showLightOn();
            timeLeft = 60;
            display.showTimeLeft(timeLeft);
        } else if (currentState == States.COOKING_STATE) {
            timeLeft += 60;
            display.showTimeLeft(timeLeft);
        }
    }
```

The methods for processing the opening and closing of the door are similar and we leave those as exercises.

10.5 An Improved Design

The above solution is our first attempt at solving this problem. We have correctly analyzed the problem and proposed an "object-oriented" solution. Our next task is to critically examine our solution to see how well it conforms to the principles of good object-oriented design. With this end in mind, we present two flaws in the above design. As it turns out, these flaws can be corrected by recognizing and applying the appropriate design patterns.

10.5.1 Complexity in `Microwave`

`Microwave` has been designed as a large class that takes care of handling all states and events. Although the methods in our class do not seem too complex, it is easy to see that in a larger system, things could easily get out of hand. In previous chapters, we have seen that complexity is caused by having a large number of conditionals in our methods. In `Microwave`, each method that processes an event has conditionals that switch on the value stored in `currentState`. In the previous chapter, we created an inheritance hierarchy to subclass the variant behavior, and succeeded in reducing the complexity of the individual methods and also in facilitating reuse. Such a subclassing is indeed possible here as well, but since a state is a common theme in any application modeled as an FSM, the idea is expressed as a design pattern called the **State Pattern.**

10.5.2 Communication among objects

In our design, objects are communicating in two contexts:

1. Events specific to the operation of the microwave, for example, the clicking of the Door Close event.

2. Events of a more general nature that could find relevance in any application. The only such event in this application is the ticking of the clock; this is clearly something that would be relevant in any time-dependent operation.

In both these cases, the `Microwave` object is the interested listener. The application specific events are being caught in the GUI and sent to `Microwave` by invoking the appropriate method. There is some coupling involved here, but since the GUI has been developed specifically for this application, this is not a serious concern. As the system operates, one should expect that changes will be needed and these will require new kinds of events to be added. In our current solution, this will require adding new methods to `Microwave`.

Consider now the more general events, which, in our example, are limited to clock ticks. We have written a class, `Clock`, that is specifically tailored for `Microwave`. Since the clock serves the same purpose in any application, we would like to have a general `Clock` class that can be instantiated wherever it is needed.

In both these cases above, what we see is that our system employs a form of communication where the entire responsibility for the communication rests with the sender, which is not desirable. In the first case, it appears to be less harmful, but as we shall see, reuse is facilitated when the responsibility is moved to the listener. In the second case, moving the responsibility to the listener helps us to define a class that can be used across possibly many applications. In general, designs like the one we had in this section make it the responsibility of the event *generator* to get hold of all the listeners and explicitly maintain a reference to each one of them. When the event occurs, every listener must be notified. This is a poor assignment of responsibilities for three reasons.

1. The event generator has to keep track of all the different classes of objects that are interested in listening, and the various ways in which they have to be notified. This makes the sender vulnerable to changes in the listener classes, which can be avoided if we have one standard format that all listeners must adhere to.

2. Responsibility for registering interest rests with the sender. This implies that when interested listeners are joining the system, the sender must somehow detect them and add them. Instead, if the listeners had this responsibility, they could simply invoke the appropriate method on the sender.

3. The set of listeners cannot change dynamically. We would like to have the flexibility that a listener object can shut off all incoming messages from a particular sender. (This would be preferable to a situation where the listener hears all messages but does not act on those from some sources.) Such a change in listener preference cannot be detected by the sender alone. If the listeners could register and de-register themselves, this would be easily accomplished.

The above drawbacks point to the fact that in a situation where the responsibility rests with the sender, we have **tightly coupled communication.** There are two standard solution frameworks for loosely coupled communication: the **Observer Pattern** and **Event-driven Communications**. We shall explore both of these in the context of our system; the observer pattern will be used for listening to clock ticks, and events will be used to transmit external events in the GUI.

10.5.3 Using the State Pattern

Using the state pattern is closely connected to the idea of modeling with FSMs. Typical situations that employ the state pattern have the following characteristic elements:

1. A collection of states, with each state being defined by distinct behavior.

2. A set of external inputs to which the system must respond.

3. A **context** in which the FSM operates.

The necessity of the first two is obvious. The third one must exist simply because the states are ephemeral and we need some "temporal glue" that provides continuity to the system. In our example, the context does not serve any purpose beyond that; in general the context may be a class that plays a much broader role and the FSM may be used to model only a small portion of the system responsibilities. The context must therefore have the following:

1. A field to track the current state of the FSM.

2. A mechanism to record the change of state.

In addition, the context may provide other mechanisms depending on the particular details of our implementation. These include, but are not limited to, the following:

1. Provide a mechanism to effect state transitions.

2. Provide methods for entities outside the system to communicate with the FSM.

3. Keep track of external entities that may need to be notified in response to internal changes.

10.5.4 Creating the State Hierarchy

In the solution given in Section 10.4, the `Microwave` class had a variable `currentState` on which the behavior was conditionally executed. This is reminiscent of the use of `bookType` in the library system in Chapter 7. Since the design in that case was improved by replacing conditionals with subclasses, we should expect to do something similar here as well with an abstract superclass subclassed by several concrete ones. The natural thing here would be to have an abstract superclass that denotes the microwave state and one concrete subclass for each of the possible actual states.

There is, however, an important difference. The library system would have numerous `LoanableItem` objects, each of which would assume the type of one of the subclasses. In the microwave, however, there is just a single microwave and rather than belong to one of the state subclasses, the microwave actually moves from state to state. As a consequence, it is more appropriate to divide the `Microwave` class into two:

1. A part that deals with state information. This forms a hierarchy formed with an abstract superclass to denote the general idea of a microwave state and one subclass for each of the actual states. This structure is shown in Figure 10.10.

2. A second part, which deals with the contextual information. We could view the original `Microwave` class as now simply holding the contextual information required for the operation of the FSM. It is therefore aptly renamed `MicrowaveContext`.

10.5.5 Transitioning between States

Transitioning to a new state is an operation that involves knowledge of the other states. Before we decide how to implement this, it is important that we examine how the information about the states and transitions is stored. We have seen that the FSM can be represented either by a transition table or in a pictorial fashion with boxes showing the states and arrows showing the transition between them. These two representations are somewhat like, but not the same as, the two standard methods for storing directed graphs: *adjacency matrices* and *adjacency lists.* In an adjacency matrix, we have a centralized storage structure. Each vertex has an associated index, and we use these indices to access the vertices and determine the connectivity between vertices. In an adjacency list, we have a more distributed storage; each vertex contains a list of the vertices which are its immediate neighbors. These two representations lead to two possible implementations for handling transitions between states.

Centralized Representation

In this approach, we first associate an index with each state. Let us assign the index 0 to the Door Closed state (the initial state), index 1 to the Door Open state, and index 2 to the Cooking state. To keep track of this mapping, we create an array; thus the context has an array of `MicrowaveState` named `states` such that `states[0]` would store a reference to the Door Closed state, `states[1]` would store a reference to the Door Open state, and `states[2]` would store a reference to the Cooking state. Some applications may designate an error state to handle unexpected conditions; in our case we might use the index 3 if we wished to do that.

Next, we work on the transitions. We know from the state transition table (Figure 10.3) that transitions may occur only when one of the events occurs. We may assign numeric values 0, 1, 2, 3, and 4 to Open Door, Close Door, Press Cook, Clock Ticks, and Timer Runs Out, respectively. Thus, the transition table can be represented as below.

1	0	2	0	0
1	0	1	1	1
1	2	2	2	0

The table is interpreted as follows. The rows correspond to transitions from a given state and the columns represent transitions when a certain input event occurs. For example, entries in the first row (indexed 0) correspond to transitions from the Door Closed state. Similarly, entries in the first column (indexed 0) show the transitions when the Open Door event occurs. We can interpret the other rows and columns in a similar way. For example, the entry at (2,4) holds the index of the state that system transitions to, when the Timer Runs Out event occurs in the Cooking state.

The Java code for setting up the table would be

```
int[][] transitions = {{1, 0 , 2 , 0 , 0},
                       {1, 0 , 1 , 1 , 1},
                       {1, 2 , 2 , 2 , 0}};
```

That was simple enough, but then the implementation would be cleaner if the event Timer Runs Out (corresponding to the last column) be externally generated. If we intend to keep `Clock` as general as possible, we need an extra class, say, `Timer`, which can be set to some value initially, then receive clock signals, and eventually send the Timer Runs Out event to `Microwave`. In addition, we might want the `Timer` object to be capable of pausing and resuming whenever cooking is interrupted and resumed.

An alternative to generating the Timer Runs Out event externally would be to have the `Microwave` itself keep track of the remaining cook time. In such an arrangement, we can treat the transition table as still valid provided that we simulate the Timer Runs Out event internally and then use it to index the table as before.

The variable `currentState` now stores an `int`, which is the index of the current state. When a state wants to relinquish control, it invokes the `changeState()` method on the context, passing the index of the event in question. The code for `changeState()` would be

```
public void changeState(int event) {
  currentState = transitions[currentState][event];
}
```

The parameter `event` would be a number that represents one of the five events and hence would be between 0 and 4.

The method determines the index of the next state by looking up the transition table. The reference to the actual state object is determined by indexing into the array `states`.

Localized Representation

In this representation, each state directly provides a reference to the next state when it has to relinquish control. The context, of course, needs to be informed of the change of state, and this is done once again using the `changeState()` method.

```
public void changeState(MicrowaveState state) {
  currentState = state;
}
```

The context stores a reference to the current state as before.

Comparison of the Approaches

Both approaches have their pros and cons. In the centralized approach, each state can be written independently of the others. The approach has the advantage that the code for a state does not have to be modified unless we want to change its behavior. The transitions can be changed and new states linked to existing ones in code external to the states. This allows a kind of reuse where we can create a library of states for a particular application domain and use these repeatedly for several applications.

In the localized approach, each state is aware of all the other states and therefore the author of each state is required to be aware of how it connects to the FSM. This approach has the advantage of simplicity in that the additional work of decoding all the exit conditions and "assembling" the FSM is not required. In situations where an FSM is being used to

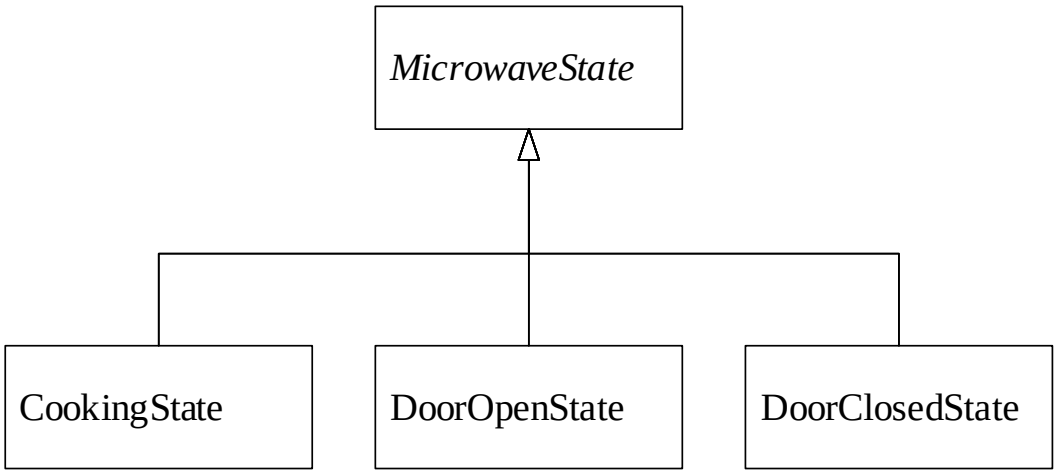

Figure 10.10: Microwave State Hierarchy: The three states in which the microwave can be are represented by three classes. Since we need to refer to the current state without actually knowing what it is, we relate them through the superclass `MicrowaveState`.

model something specific like, say, an algorithm for a specialized communication protocol, it is unlikely that a library of states can be reused across the domain. In that case, it may be beneficial to use the localized approach and embed the transition information within each state. We shall implement the transitions for our case-study using the localized approach.

10.5.6 The State Classes

We now elaborate on the state classes. The abstract class, `MicrowaveState`, contains methods to handle the various events and its class diagram is shown in Figure 10.11. The meanings of most of the methods should be obvious. When the microwave changes state, some actions may need to be carried out such as variable initializations and communication with external agents. It is convenient to have all of these actions executed in a method, which we term `enter()`. Similarly, while leaving a state, some cleanup might be necessary, which calls for the `leave()` method. These two are declared `abstract`. The other five methods specify the actions for the five possible events.

The display needs to be updated as state transitions occur. We need to determine what actions should be carried out as the microwave enters and leaves the different states. While there is more than one way of assigning responsibilities to the `enter()` and `leave()` methods, a close examination of the states shows that the following would work.

1. When the `DoorOpenState` is entered, the status of the door should be shown as open and when leaving the state, the door status should be marked as closed.

2. At the time of entering the `DoorClosed` state, the light status should be displayed as turned off and while leaving the state, the light status should be shown as turned on.

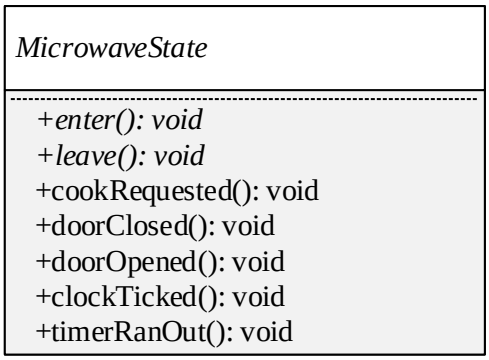

Figure 10.11: Class Diagram for `MicrowaveState`. The `enter` method is a place holder for doing the necessary processing while entering this state. The `leave` method specifies what is to be done while leaving the state. The other five methods specify the actions for the five possible events and will do nothing as the default action.

3. As the state transitions to the `Cooking` state, the timer should be activated, the time left for cooking should be shown as 60 seconds, and the microwave status should be shown as cooking. While leaving the state, the timer should be turned off, the time left should be displayed as 0, and the microwave status should be shown as not cooking.

This means that every subclass of `MicrowaveState` should override the `enter()` and `leave()` methods.

Next, we develop the the class diagrams for the individual states. When the door is open, the microwave does not respond to anything other than the Door Close event. Therefore, in the class diagram for `DoorOpenState` (Figure 10.12), we only show the methods `doorClosed()`, `enter()`, and `leave()`. Similar interpretations can be made for `DoorClosedState` and `CookingState`, which are respectively shown in Figures 10.13 and 10.14.

10.5.7 Timing the Cooking

In our initial implementation, the `Microwave` object kept track of the cooking time. This simple approach overlooked the fact that a timer is an object in its own right. Creating a separate `Timer` class abstracts out the functionality needed to set a timer value and be notified when the timer runs out. This simplifies the construction of the FSM and helps with the construction of more complicated systems should we wish to employ multiple timers, perhaps one for cooking, and the others for setting alarms. (Some real-life microwaves do have more than one timer.)

An obvious consequence of the decision to introduce a timer is that the `Timer` object, and not `MicrowaveContext` would now become the client of `Clock`, which must notify the `Timer` in a loosely coupled manner.

Loosely coupled communication must have three properties:

1. The listener is responsible for registering interest.

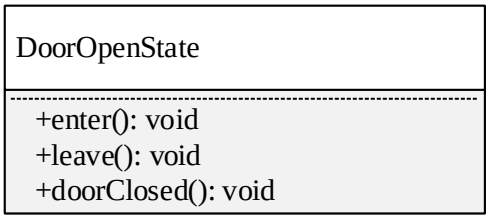

Figure 10.12: Class Diagram for `DoorOpenState`. Only the methods that override the superclass's methods are shown here. The `enter()` method would make the display show that the door is open. The `leave()` method would make the display show that the door is closed. The only event this state reacts to is the Door Close event.

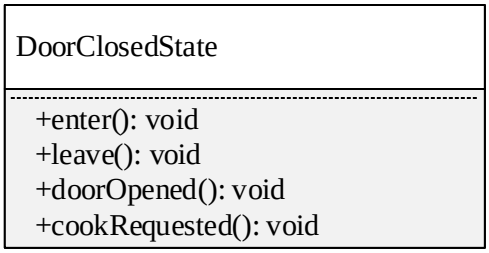

Figure 10.13: Class Diagram for `DoorClosedState`. Only the methods that override the superclass's methods are shown here. The `enter()` method would make the display show that the light is turned off. The `leave()` method would make the display show that the light is turned on. The only events this state reacts to are the Door Open and Cook Request events.

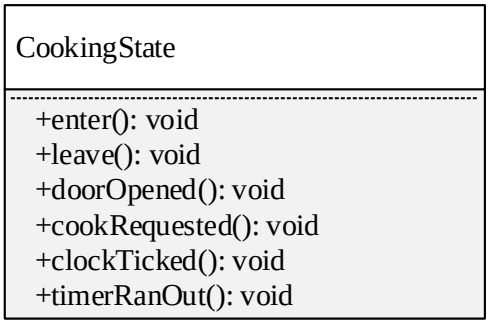

Figure 10.14: Class Diagram for `CookingState`. This class overrides all methods of the superclass except `doorClosed()`. The `enter()` method would initialize the cooking process by setting up the timer. It would also display the time remaining for cooking as 60 seconds and display that the microwave is cooking. The `leave()` method would make the display show that the microwave is no longer cooking and that the remaining cook time is 0.

2. All interested listeners share some common interface so that the sender need not distinguish between listeners.

3. The sender has a mechanism for maintaining a collection of the interested listeners.

The **Observer Pattern** gives us a mechanism that makes this possible.

The Observer Pattern

There are two categories of players in the Observer Pattern:

1. The **observable**, which is usually a single object, and

2. The **observers**, of which there may be several. It is the responsibility of the observable to provide a method by which the observer can register interest. Once an observer has been registered, it is the responsibility of the observable to notify that observer of any state changes within the observer. In order to accomplish this without causing tight coupling, every observer must have a method with a signature that has been agreed upon.

The observer pattern can be implemented using a JDK class called `PropertyChangeSupport` and a JDK interface named `PropertyChangeListener`. A class could have its state described by multiple aspects called **properties**. A **Java bean** is a Java class that has a public no-arg constructor, implements the `Serializable` interface, and maintains properties by storing them in private fields with public getters and setter methods. A `PropertyChangeListener` may register itself to observe changes to one or more properties in a Java bean[2].
`PropertyChangeSupport` has the following categories of methods:

1. In the first category, we have methods for maintaining the list of observers. The method `addPropertyChangeListener(PropertyChangeListener listener)` adds the given listener to this list. There are two ways of deleting observers. An observer can be deleted by using the method `removePropertyChangeListener(PropertyChangeListener listener)`.

2. There are several variants of a method named `firePropertyChange()`, which allows the notification of the `PropertyChangeListener` objects of "noteworthy" events occurring within the observer. One specific signature is

   ```
   firePropertyChange(String propertyName, Object oldValue, Object newValue)
   ```

   which informs registered observers of a change to a property identified by `propertyName` within an observable. The old and new values are to be passed as well. The caller is to indicate no specific property by passing `null` values for all three parameters.

3. The third group has a few methods to get information about the observers.

Every class that wishes to be an observer implements the `PropertyChangeListener` interface, which has the method `propertyChange()` with the following signature:

[2]We do not implement observables as Java beans in this book.

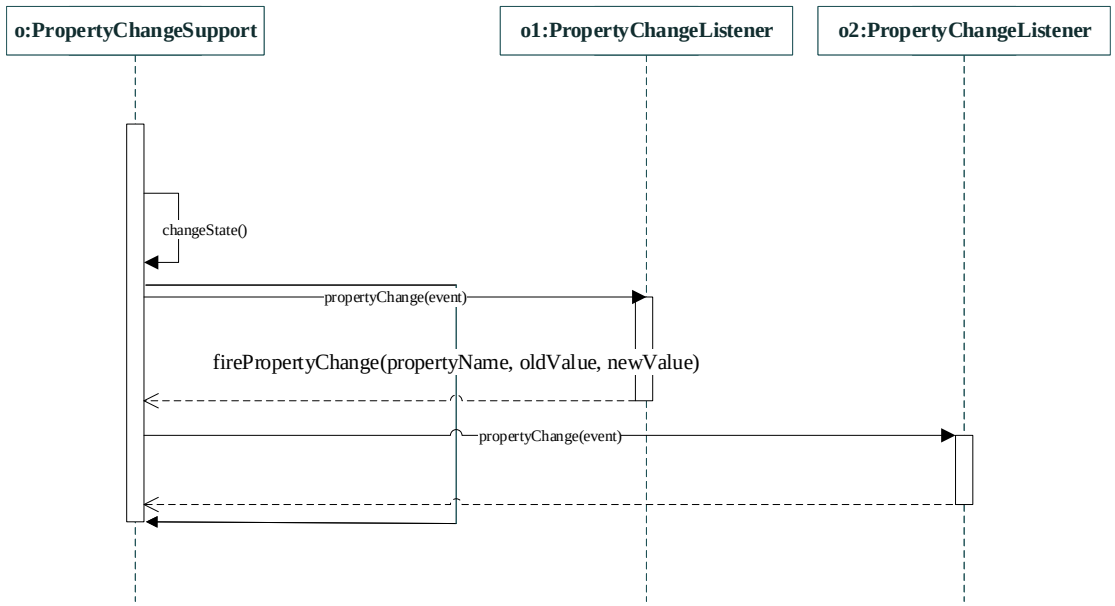

Figure 10.15: Sequence Diagram for notifying observers. When an event occurs, say, due to the execution of the `changeState()` method in the observer, the observer invokes the `firePropertyChange()` method, which calls the `propertyChange(event)` method on each of the observers registered with the `PropertyChangeSupport` object.

```
public abstract void propertyChange(PropertyChangeEvent event);
```

When `firePropertyChange()` is invoked in the `PropertyChangeSupport` object, the `propertyChange()` method is invoked once for each item in the list of "interested observers." The `PropertyChangeEvent` object has methods that can be used by the listener to determine the property name and the old and new values of the property.

The interaction between `PropertyChangeSupport` and two `PropertyChangeListener` objects is depicted in the sequence diagram in Figure 10.15. The picture shows an event occurring in the `PropertyChangeSupport`, which calls the `firePropertyChange()` method. In response to the `firePropertyChange()` method call, the `PropertyChangeSupport` object calls the `propertyChange()` methods of the two observers supplying them with information about the event(`event`).

Generating and Handling Timing Events The `Timer` class must have the following functionality.

- *The timer must have a client.* The purpose of a `Timer` object is to inform a client object `Client` about timing signals. The `Timer` object is dedicated to working with `Client`.

- *The timer must have a time period set at construction time.* We should be able to create a timer that counts down to 0 from an initial value given at construction time.
- *At the end of the time period, the timer must notify its client.*

In addition, it would be desirable to have the following functionality in a `Timer` class.

- *The timer should should notify its client whenever a certain period has elapsed.* For the Microwave, we need notifications every second, but in general, one could have this value specified by the creator at construction time or even changed periodically.
- *It should be possible to pause and resume the timer.* If we wish to modify the Microwave application so that we could resume cooking after opening the door, this feature would be very convenient.

An issue that remains is how the communication between `Timer` and its client should take place. We have at least two options.

- They communicate using the Observer Pattern with `Timer` being the observable and the client being the observer.
- `Timer` calls specifically-named methods of the client to notify it of the timer running out.

The advantage of using the Observer Pattern is that it uses a well-known paradigm and is readily understood. If we make the assumption that a timer can have multiple clients, then this approach is justified.

In this problem though, we make the assumption that a single timer will have only one client. If that is the case, the Observable Pattern would be an overkill: we do not wish to incur the overhead of maintaining a `PropertyChangeSupport` and its apparent overhead of maintaining a list of observers and a loop to notify the observers.

Even if we assume that a timer may only have one client, it is quite possible that there could be multiple timers each having its own client. In that case, it is, however, not desirable to use specifically-named methods in the client for notifications. For the `Timer` class to be more widely useful, we should have potential clients implement a common interface, which we term `Notifiable`. The functionality is given in 10.16.

The `Timer` class diagram is shown in Figure 10.17. While instantiating, the creator must supply the client object and the initial time value. The time could be added to or the timer could be stopped via the appropriately named methods. Obviously, one could add to the functionality with obvious choices including methods to pause and resume the timer.

Creating the Timer Object We now need to address the question of who should assume the responsibility of creating and managing the `Timer` object. Two options come to mind.

- **Option 1:** *The* `MicrowaveContext` *instance can create a* `Timer` *object.* The notifications would be sent to `MicrowaveContext`. `MicrowaveContext` fowards these notifications to the current state, which can process or ignore that message.
- **Option 2:** *The* `CookingState` *class itself create the* `Timer` *and receive the signals directly.*

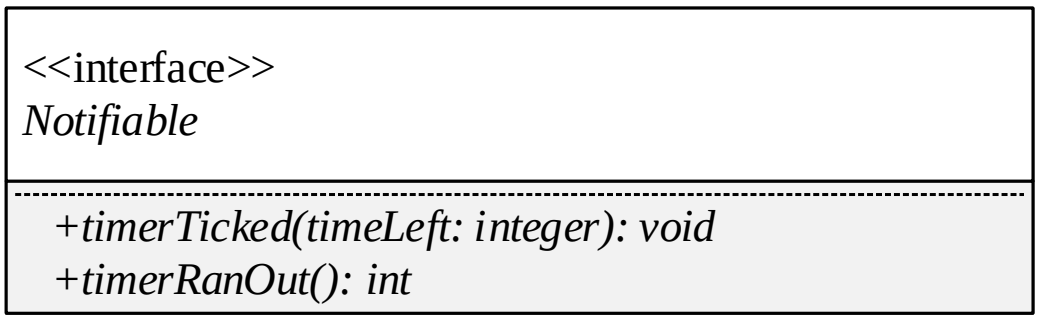

Figure 10.16: Interface `Notifiable`. Generally speaking, any object may need to create a timer. For a timer to report back to its creator, it then becomes necessary to have an interface the timer's creator can implement, so it can be appropriately notified.

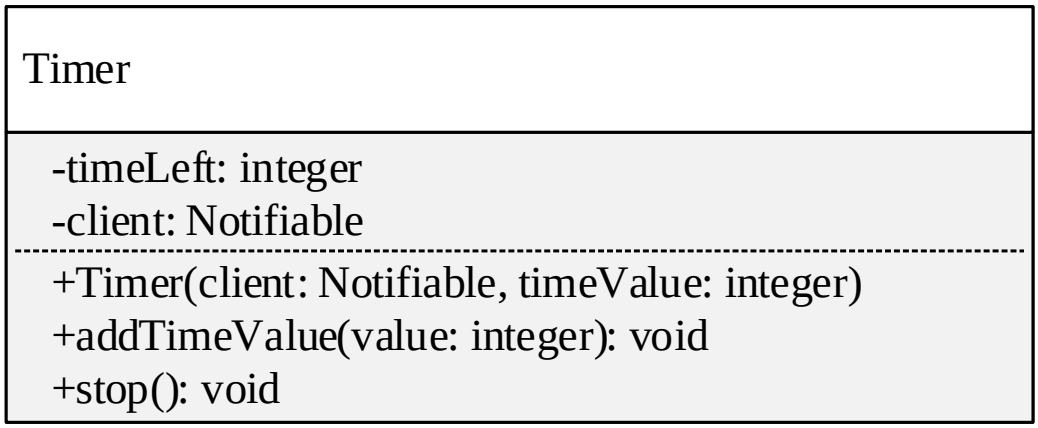

Figure 10.17: The `Timer` class. The class is instantiated by a client, which implements the `Notifiable` interface. The client's id is maintained by the timer, so it can notify the client. One could add more methods for example, to pause and resume the timer.

The first approach has the advantage that it provides a uniform approach to handling external events. Notifications from `MicrowaveDisplay` and `Timer` are all sent to `MicrowaveContext`, which delivers them to the currently active state.

The second approach, however, recognizes the fact that the `Timer` is needed by the `CookingState` class, and makes `CookingState` responsible for managing its needs. This avoids the unnecessary coupling between `CookingState` and `MicrowaveContext` that would otherwise result, and avoids some difficulties that might occur in system enhancements (see below).

A problem with having the context as the `Timer`'s client is that it makes the design not very extensible in certain situations. Assume that in the future, we want to provide several pre-determined cooking schemes: for example, we might have a specific scheme for boiling a cup of water and another for reheating frozen pasta. Perhaps the former might involve using the power tube for 100 seconds at 100 percent power and the second cooking scheme might cook for two minutes at 50 percent of the maximum power level followed by one minute at 100 percent power. Since such multiple schemes have different behaviors, we would have multiple state classes as well. The `MicrowaveContext` would then be the client of the timers for all such new classes as well, and would ultimately have more complicated data structures to keep track of these timers and significantly more complicated interactions with the state classes. It would certainly be simpler if each state class managed all the timers it needs.

Microwave Context

In the design of the `MicrowaveContext` class, we must address the question of how the concrete state classes perform the necessary computations. Some of the actions may be purely local to the state, and these can be handled in an obvious way. There are two kinds of actions that have an effect outside the state: *(i) actions that require a change of state,* and *(ii) actions that require making changes to an entity outside the FSM.*

In a typical FSM, each state has some impact on the environment in which it is operating. In our case, these are actions that change the display. For instance, when the cook button is pressed while in the cooking state, the number of seconds remaining should be increased by 60. The question here is how to implement the communication from the cooking state to the display object. It should be no surprise that we have more than one option for handling these.

- **Option 1:** *All communication goes through the context.*
- **Option 2:** *Each state communicates independently with the external entities.*

The first option is appropriate in situations where we want the entire FSM subsystem to be a unit that can inter-operate with several environments. For instance, we may decide that we are no longer having a simple display to show what is going on, but want to manipulate device drivers that actually turn the light and power tube on and off. Such a change could be accomplished by changing just the context; if we had chosen the second option, every state would have to be changed to communicate with the new external environment. Note that Option 1 requires that the context provide methods for communication that can be invoked by the states. This would result in an additional overhead of a method call.

In the second option, each state must keep track of the concrete entities that it wishes to communicate with and has to be tailored to that interface. This clearly makes the state dependent on these interfaces and thus introduces some additional coupling. This seems

to suggest that Option 1 is always preferable, which would not be correct for the following reason. Consider a situation where each state has some distinct kinds of external entities which it communicates with; *if all the communication went through the context, the context would have to provide methods for each kind of entity, make it a very unwieldy class.* In such a situation it is preferable that each state communicate directly with its external clients. The coupling that may result can be significantly reduced if all the external entities were to be implementations of stable abstractions.

The class diagram for `MicrowaveContext` is shown in Figure 10.5.7. In our design all communication between the states and the UI goes through the context. It is convenient to have in the context a method `setDisplay()` to set the reference to the display object. The current state can be changed by executing the method `changeState()`.

Figure 10.19 shows the sequence of calls that takes place when the clock ticks.

Implementation Using the State and Observer Patterns

We begin showing salient portions of our implementation with the major changes in `Clock`. The class maintains a]`PropertyChangeSupport` object to keep track of the observers, and delegates calls to add the observers to this object. After each clock tick, the `firePropertyChange()` method is called to inform the observers of a clock tick. Note that there is no real property involved, so we follow the recommendation of the JDK and pass `null` values for the property name the new and old values.

```
public class Clock implements Runnable {
  // several fields and methods not shown
  private final PropertyChangeSupport propertyChangeSupport = new PropertyChangeSupport(this);
  public void addPropertyChangeListener(PropertyChangeListener listener) {
    this.propertyChangeSupport.addPropertyChangeListener(listener);
  }

  public void run() {
    try {
      while (true) {
        Thread.sleep(1000);
        this.propertyChangeSupport.firePropertyChange(null, null, true);
      }
    } catch (InterruptedException ie) {
  }
}
```

The `Timer` class maintains fields to keep track of the time and the client. Values to both fields are passed at creation time, so that sets up the timer for a certain client with an initial time value. The timer then adds itself as an observer of the clock. The `addTimeValue()`, which changes the timer value is quite straightforward.

Each clock tick results in a call to the `propertyChange()` method. The timer decrements the time counter and checks if it has reached 0. Depending on this, one of the two methods, `timerTicked()` or `timerRanOut()`, is called. Once the timer has run out, the timer takes itself out as an observer of the clock.

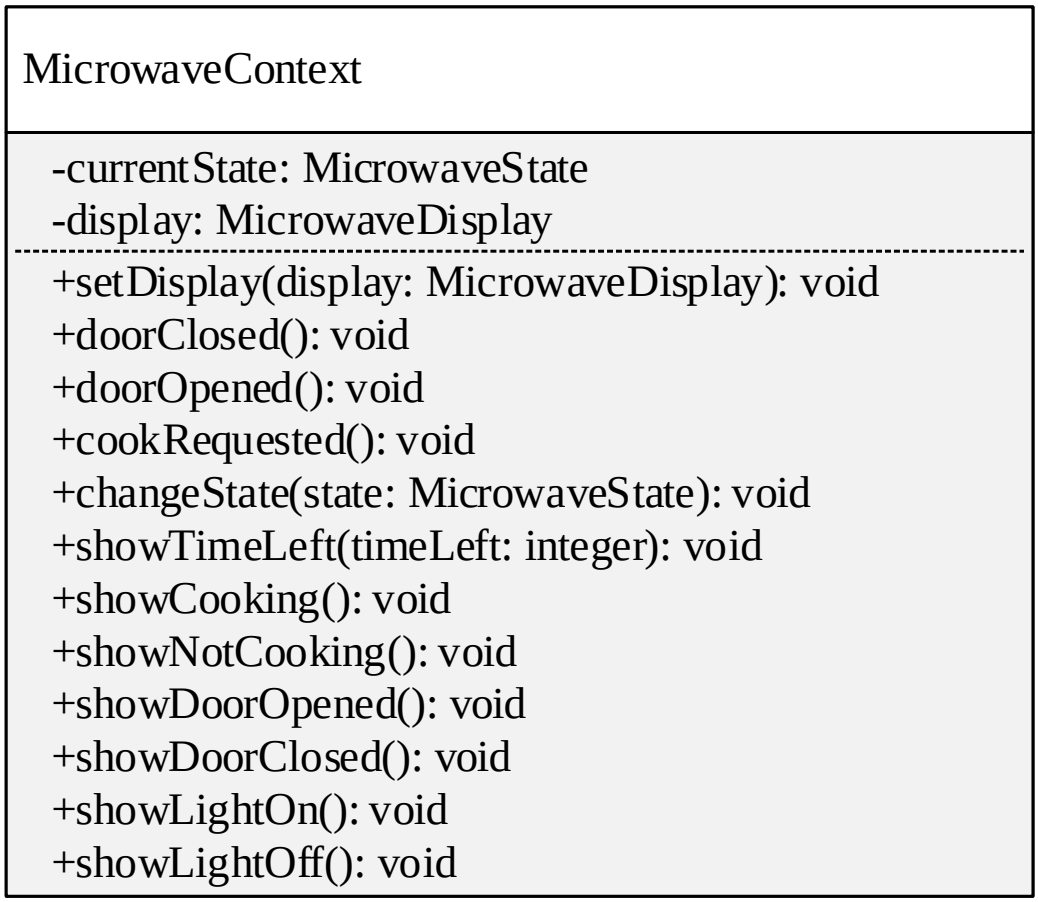

Figure 10.18: Class Diagram for `MicrowaveContext`. There three event methods, one for each event that comes from the GUI. Then there are the methods called by the state classes to change the display. The `changeState()` method is called to change the state of the FSM.

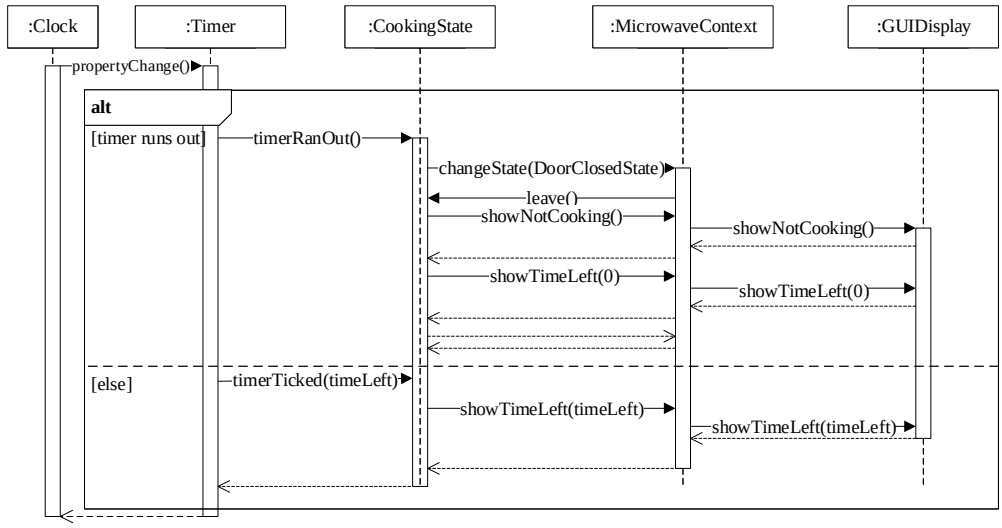

Figure 10.19: The sequence of actions that takes place when the clock ticks. The `Clock` object notifies the `Timer` object. Depending on whether the timer runs out or not, the `CookingState` class issues different messages to `MicrowaveContext`. Notice a sequence of dotted lines that show the replies to the `leave()` and `changeState()` method calls.

```
public class Timer implements PropertyChangeListener {
  private int timeValue;
  private Notifiable client;
  public Timer(Notifiable client, int timeValue) {
    this.client = client;
    this.timeValue = timeValue;
    Clock.instance().addPropertyChangeListener(this);
}

  public void addTimeValue(int value) {
    timeValue += value;
  }

  public void propertyChange(PropertyChangeEvent arg0) {
    if (--timeValue <= 0) {
      client.timerRanOut();
      Clock.instance().removePropertyChangeListener(this);
    } else {
      client.timerTicked(timeValue);
    }
  }
}
```

Moving onto the `MicrowaveContext` class, we note that the `changeState()` method calls the `leave()` method of the current state before switching to the new state and initializing it.

```
public void changeState(MicrowaveState nextState) {
     currentState.leave();
     currentState = nextState;
     currentState.enter();
}
```

Some representative methods of `MicrowaveContext` are given below. These are fairly straightforward methods.

```
public void doorClosed() {
   currentState.doorClosed();
}

public void showTimeLeft(int time) {
   display.showTimeLeft(time);
}
```

The `enter()` and `leave()` methods of the `CookingState` are given below. The `enter()` method creates a timer and modifies the display, whereas the `leave()` method sets the reference to the `Timer` to `null`, so it can be garbage collected.

```
    public void enter() {
        timer = new Timer(this, 60);
        MicrowaveContext.instance().showCooking();
        MicrowaveContext.instance().showTimeLeft(timer.getTimeValue());
    }

    public void leave() {
        timer.stop();
        timer = null;
        MicrowaveContext.instance().showNotCooking();
        MicrowaveContext.instance().showTimeLeft(0);
    }
```

`MicrowaveState` has default methods for processing each of the events, and the concrete state classes are supposed to override the appropriate ones to implement the required functionality.

```
public abstract class MicrowaveState {
    public abstract void enter();
    public abstract void leave(MicrowaveState nextState);
    public void cookRequested() {
    }
    public void doorOpened() {
    }
    public void doorClosed() {
    }
    public void timerTicked() {
    }
```

The Observer Pattern
Where do we employ this? An abstraction has several parts, which have to be encapsulated separately, but there is a need to maintain communication between them.
What problem are we facing? Communicating with an object essentially requires that some method of the object be invoked. This implies that the object initiating the communication has to know which method of the recipient(s) is to be invoked. If the sender must keep track of the methods of all recipients, we get tight coupling between the various parts of the abstraction.
How have we solved it? We place the responsibility for communication on the receiver instead of the sender. The sender (also known as the Subject or the Observable) has a method that allows recipients (Observers) to register their interest with the sender. All the recipients of the message implement a common method (or an interface) for receiving messages. When there is a need for communication, the sender invokes the common method on all the receivers that have registered their interest with the sender.

Figure 10.20: The Observer Pattern

```
    public void timerRanOut() {
    }
}
```

10.6 Eliminating the Conditionals

In the design presented above, the `handle` method in `GUIDisplay` handles the clicks on the buttons. This conditional switches on the kind of event, which in turn is decided by external input.

```
public void handle(ActionEvent event) {
    if (event.getSource().equals(doorCloser)) {
        microwaveContext.doorClosed();
    } else if (event.getSource().equals(doorOpener)) {
        microwaveContext.doorOpened();
    } else if (event.getSource().equals(cookButton)) {
        microwaveContext.cookRequested();
    }
}
```

The other source of events is the `Timer` class.

In a more complicated system, the number of events and states could be quite large and there could be numerous sources of events. Having the events correctly delivered to the correct state could be quite an error-prone process.

Concrete vs Abstract entities in Design Patterns

Often in Design Patterns, we find both an abstract and a concrete version of an entity. Sometimes the abstract entity is present just as a type (interface) and at other times, it is an abstract class. In the Visitor pattern, we encountered the use of interfaces. Interfaces suffice in situations where all the required properties of the entity can be expressed without any implementation.
In the State pattern, the abstract state class allows us to capture default behavior and also store references to other entities. In our microwave implementation we do not have an abstract context. However, in an implementation where the state transitions are stored as a table, an abstract context to help with reuse.
In the Observer pattern, the Observable is an abstract class. The mechanism used to enforce the pattern requires that the Observable store references to the Observers, and notify them as needed. If no implementation is provided, we may end up in a situation where the correctness of the pattern is compromised. Since the Observer entity on the other hand needs just the `update` method with no restrictions on its behavior, it is left as an interface.
Another important benefit of the abstract entities is that they enable some form of type checking. The abstract Observer class maintains a polymorphic container to keep track of all observers and the abstract entity helps ensure that all these objects have implemented the `update` method. In the absence of this, Observable would be storing a collection of objects and strong typing would not be possible.

Figure 10.21: Abstract Entities in Patterns

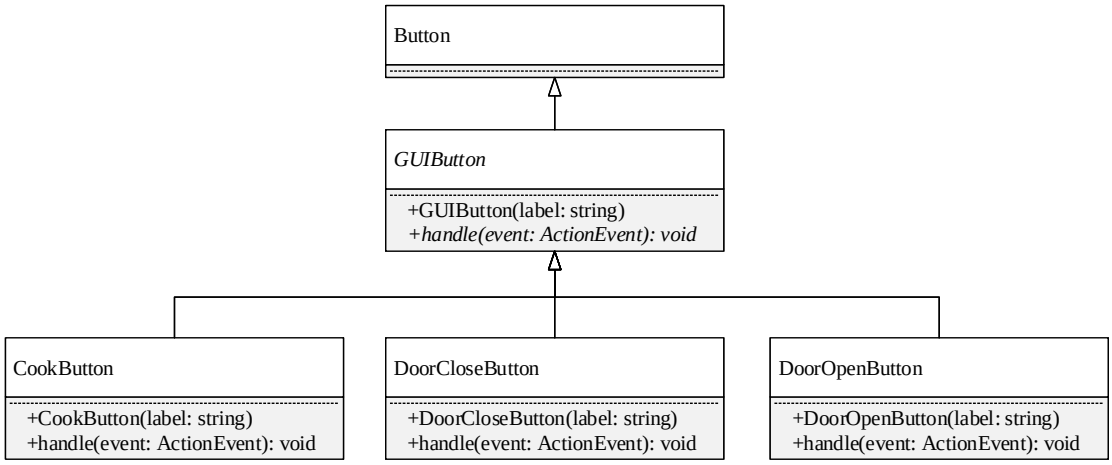

Figure 10.22: Button Hierarchy: The three concrete `Button` classes represent the three buttons in the interface. Each button handles its own clicks by calling an appropriate method of the context.

In this section, we describe a design strategy that practically eliminates these issues. The approach has the following characteristics.

- It eliminates most of the conditionals in the system. This is based on the refactoring rule of replacing conditionals by polymorphism.
- To some extent, the scheme is a mechanical process based on the state-transition table (or diagram).
- The approach increases the number of classes. While this is a drawback, the classes are quite simple and small and the code is quite obvious.

The major conditionals in the previous solution to the system appear in the `handle()` method, which switches to different method calls based on the button that was clicked. This method is switching on the type of button. We can eliminate the conditionals by replacing the conditionals by polymorphism.

As we know, replacing conditionals by polymorphism requires the creation of a subtype for each leg of the conditional. Here it translates into the creation of a hierarchy of `Button` classes.

This hierarchy is shown in Figure 10.22. The common superclass `GUIButton` is abstract and should help us treat all buttons in a uniform way, should that be necessary. In its constructor, it also makes every subclass listen to its own clicks.

Notice that the `timerTicked()` method in `MicrowaveContext` is unique among that class's methods that listen to events, in that it has a parameter. In general, event handling methods may need to have multiple parameters. For example, when a mouse is clicked in a drawing program, the event listener would need to know the x and y coordinates of the clicked point. For the sake of uniformity, it is perhaps advisable for every listener method to have a parameter that describes the event.

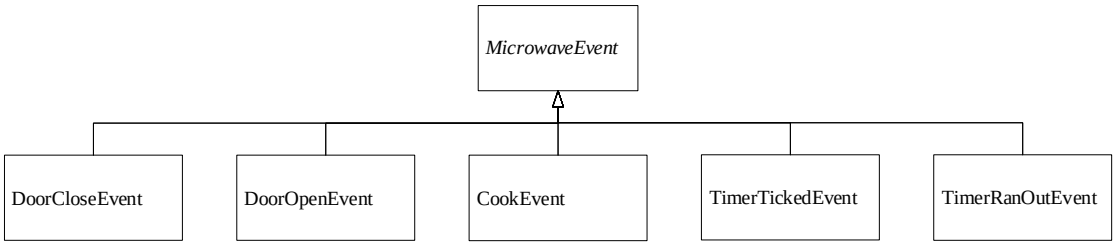

Figure 10.23: Event Hierarchy: For every event in the system, we create one class. An event object provides a convenient mechanism to hold together all the information needed to describe the event.

This means that we might wish to have a separate class for every possible event in the system. To treat these events in a polymorphic manner, it is also advisable to have a common superclass for all these events. For the microwave, the hierarchy would appear as in Figure 10.23.

Now that we have an event class for every possible event in the system, we can do some refactoring to simplify the names associated with the listener methods in the context and the state methods. Rather than have method names such as `doorOpened()` and `cookRequested()`, we can have the same name `handleEvent()` for all listeners, and give each function a unique signature by having a parameter of the event type.

To see the interaction among the classes in the hierarchy, let us assume that currently the current state is `DoorClosedState` and that the button to open the door is clicked. The sequence of actions that takes place is as follows:

1. The user clicks the Cook button. This generates an instance of `ActionEvent` and control goes to the `handle()` method of `CookButton`.

2. The `handle()` method creates an instance of `CookRequestEvent`. It then sends the event as parameter in a call to the `handleEvent()` method of `MicrowaveContext`.

3. In the `handleEvent()` method of `MicrowaveContext`, the parameter is used to call the `handlEvent()` method of `DoorClosedState`. This is another polymorphic call.

4. The `handlEvent()` method of `DoorClosedState` changes the current state by calling the `changeState()` method of `MicrowaveContext`, which in turn modifies the display through calls to the `leave()` method of `DoorClosedState` and the `enter()` method of `CookingState`.

10.6.1 Implementation

We have one class for each of the events. The class `MicrowaveEvent` serves as the superclass of all. Here is the code for `TimerTickedEvent`, which is the only non-singleton class.

```
public class TimerTickedEvent extends MicrowaveEvent {
    private int timeLeft;
```

```
    public TimerTickedEvent(int value) {
        this.timeLeft = value;
    }
    public int getTimeLeft() {
        return timeLeft;
    }
}
```

When the clock ticks, as before the `propertyChange()` method of `Timer` gets called. If the timer goes to 0 or below, the (only) instance of `TimerRanOutEvent` (a singleton) is used as a parameter to call the client's (`CookingState`) `handleEvent()` method. Otherwise, an instance of `TimerTickedEvent` is created with the amount of time left for cooking stored in it.

```
public void propertyChange(PropertyChangeEvent arg0) {
  if (--timeValue <= 0) {
    client.handleEvent(TimerRanOutEvent.instance());
    Clock.instance().removePropertyChangeListener(this);
  } else {
    client.handleEvent(new TimerTickedEvent(timeValue));
  }
}
```

As a representative piece of code, let us take a look at the `CookButton` class's code. When this button is clicked, the `handle()` method springs into action, calling the context's `handleEvent()` method for processing cook requests. Notice how Java helps in selecting the appropriate method. The context then forwards the request to the current state object.

```
public class CookButton extends GUIButton implements EventHandler<ActionEvent> {
    public CookButton(String string) {
        super(string);
    }
    public void handle(ActionEvent event) {
        MicrowaveContext.instance().handleEvent(CookRequestEvent.instance());
    }
}
```

Finally, the state classes are the same as before, except for the fact that the names of the event handling methods have been changed.

10.7 Employing the FSM Model for Other Types of Applications

The above discussion should be convincing enough to realize that the FSM model is quite appropriate for solving problems related to devices. In fact, the approach could be successfully employed to analyze, design, and implement other problems that involve states. We outline two such domains in this section.

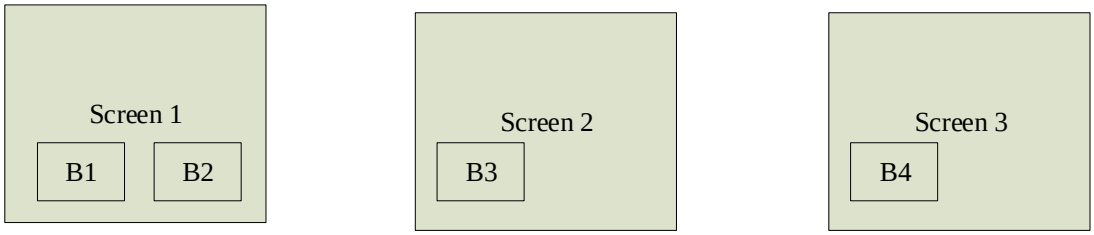

Figure 10.24: A simple GUI application with three screens. Screen 1 has two buttons, and the other two have one button each. Clicking on B1 causes the GUI to switch to Screen 2, clicking on B3 causes a switch to Screen 3, and clicking on B4 makes the GUI show Screen 1. A click on B2 closes the program. The point is that this can be analyzed and solved using the FSM approach.

10.7.1 Graphical User Interfaces (GUIs)

A GUI application typically involves many screens, each having multiple widgets such as textboxes, scrollbars, buttons, etc. Using some of these widgets (for example, clicking a button), sometimes presents a different screen.

For a little more specificity, consider the following scenario (also depicted in **??**). The GUI has three screens (Screen 1, Screen 2, and Screen 3) and shows exactly one of the three screens at any given time.

1. Screen 1 has two buttons, B1 and B2. Clicking on B1 causes the GUI to display Screen 2 and clicking on B2 closes the program itself.

2. Screen 2 has a single button, B3. Clicking on it results in Screen 3 being displayed.

3. Screen 3 has a single button, B4. Clicking on it results in Screen 1 being displayed.

We can think of each of the three screens as one of the states of an FSM and the clicking of the buttons as events. We can model the application as an FSM and come up with a state transition diagram as shown in Figure 10.25. Note the following aspects.

- To avoid complexity, we haven't shown events that cannot occur: for example, in the state `Screen1`, button B3 cannot be pressed; we could complete the diagram by a transition to the same state for such impossible events.

- The terminated state does not correspond to an actual software class. It is inserted only to show that when button B2 is pressed, the system should exit.

It is fairly straightforward to design and implement the system from the state transition diagram.

An argument can certainly be made that the above scenario is extremely simplistic, but the point here is that there is a way to apply the FSM modeling approach to construct GUI programs. We will have a far more solid example constructed in Chapter 9.

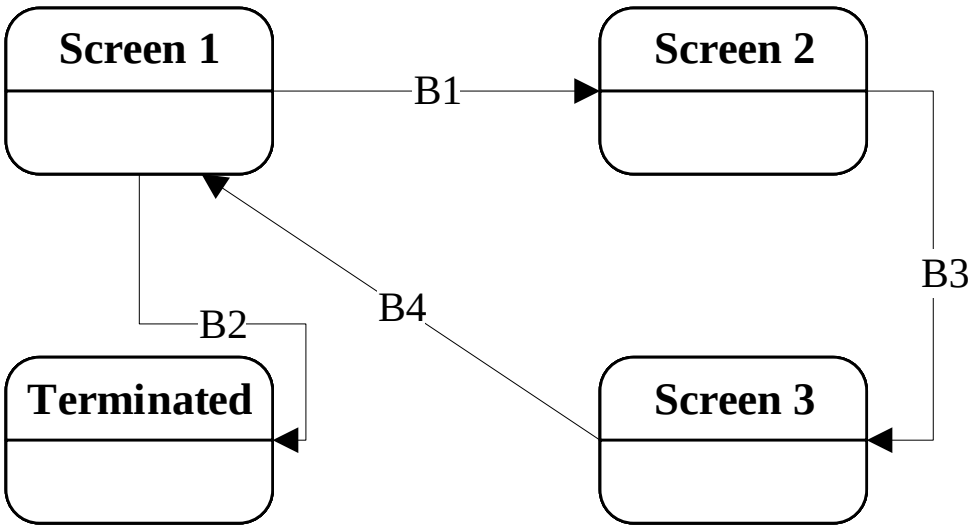

Figure 10.25: The state transition diagram for the GUI application. We have added a fourth state to take care of the program's termination. To avoid complexity, we haven't shown events that cannot occur: for example, in the state `Screen1`, button B3 cannot be pressed; we could complete the diagram by a transition to the same state for such impossible events.

10.7.2 Network Protocols

As another example of an application, consider a file-transfer program, which allows the downloads (uploads) of files from (to) a remote computer. Such a program could be thought of as consisting of a set of states as outlined below.

1. The start state. A user may may supply the user id and password of an account he/she holds on a remote machine and the IP address of that machine to log in. Assuming these are valid and a connection can be established, the program moves into the logged-in state. (See 2 below.) The program allows a command to exit.

2. The logged-in state. The user may supply the name of a remote file F_r, the name of a local file F_l, and choose either upload or download. The program goes into the file-transfer state. (See 3 below.) The program allows a command to log out, which moves it to the start state.

3. The file-transfer state. The program uploads or downloads the file. Once the file transfer is complete, the program goes back to the logged-in state.

Again, the description is quite simplistic: a real file-transfer program would be much more versatile, allowing transfers of files from multiple directories, handling errors, and so on. But those can be handled by adding more events and states to the analysis.

10.8 Discussion and Further Reading

10.8.1 Implementing the State pattern

The State pattern can be implemented in different ways, and the particular implementation that we choose depends on the role played by the context and the kind of relationship we have between the states. At one end of the spectrum, we have an implementation where the context is nothing but a repository for shared information. This is the approach recommended in [**?**]. When a state has completed all of its actions, it passes control to the next state along with a reference to the context object. We have two choices as to how this can be realized. One approach is to create a new instance of a state whenever a state terminates. Another approach is where each state is a singleton and the current state invokes `instance` to obtain a reference to the next state. In such an implementation, there is some coupling between the individual states and requires that each state be fully responsible for listening and responding to external events.

At the other end of the spectrum, we have a situation where each state is completely unaware of the existence of other states. This is accomplished by the current state terminating with a call to the context, which looks up a transition table to decide what the next state should be [**?**]. The context in this implementation provides methods to add states and transitions, which populate the transition table. These methods are accessible from code outside, which allows the context to be reused in other applications. Such an approach is particularly suitable for designing multi-panel interactive systems, such as a GUI for an application [**?**]. The approach that we have taken in the design of the microwave lies somewhere in the middle of this range.

10.8.2 Features of the State Pattern

The examples in the chapter should be illustrative enough that the following salient aspects of the State pattern can be appreciated.

1. An application can be in one of many states, and its behavior depends on the state it is in. In our example, the microwave object can be in one of three possible states: Cooking, Door Open, and Door Closed.
2. We create one class per state. We may choose to put their common functionality in an abstract superclass or make all of these states implement a common interface, so they all conform to some common type.
3. One instance of each state class is created. In the case of the Microwave, the three classes corresponding to the three states are all singletons. This way, unnecessary object creation and deletion are avoided.
4. There is a context that orchestrates the whole show. This object remembers the current state and any shared data.
5. Exactly one state is active at any given time. The context delegates the input event to the state that is currently active and therefore only the active state responds to events.
6. When an event that requires a change of state occurs, we determine the next state, which then becomes active. For example, this transfer of control occurs for the Microwave application by having each state determine the next state and then calling the `changeState` method of the context.

The mechanism for deciding the next state can be done in one of two ways.

(a) One approach would be to have a centralized controller that uses the matrix (see Section 10.5.5) to decide what the next state is. In this technique, after responding to an event, the state can return the input event to the controller, which can use the current state and the input event to determine the next state.

(b) The second approach is to have the current state determine the next state. We used this approach in our Microwave example.

The advantages of using the pattern are:

1. There is no longer a need to switch on the state in order to decide what action needs to be taken. Instead, we polymorphically choose a method to be executed.

2. New states can be added and old states reused without changing the implementation. For example, in the Microwave example, we can resume cooking after an interruption by simply having a new version of the class `CookingState`. (The reader is encouraged to make this modification.)

3. The code is more cohesive. Each state contains code relevant to it and nothing else. Only events that are of interest in this state are processed.

10.8.3 Consequences of Observer

The simplicity of the Observer pattern belies the power it conceals. In essence, we have allowed an arbitrary object to be registered as a listener, and the observable invokes a method (viz. `update`) provided by the observer. Likewise, an object can become a listener to an arbitrary number of classes. As one should expect, such power brings along a lot of caveats and consequences.

For a start, we have the problem of memory leaks. In a system that provides automatic garbage collection, objects for which no references are maintained can be cleaned up during garbage collection. If an observable stores a reference to an object it is tricky to decide when an object is no longer needed and some explicit mechanism may be needed to signal the end of an object's lifetime. Next we have the problem of the order in which observers are notified. The pattern itself does not specify any order, and if any temporal ordering is desired, explicit mechanisms, such as introduction of intermediaries, may be needed [**?**].

Since any arbitrary object can become a listener, we may end up in situations where an update method invoked by the observable has unsafe code, say for instance, an unhandled exception or a delay. The standard approach to avoid this is for the observable to have every observer on a separate thread. This solution in turn leads to other caveats for programmers, such as not registering listeners from within constructors and not adding new listeners when existing ones are being notified [**?**].

A class that listens to several observables can end up with an `update` method that is quite complex. The order in which an observer deals with notifications from observables can change the result of the computation. Two such problems, viz., *Cyclic Dependencies* and *Update Causality* are discussed in [**?**].

Computation involving threads has its own share of pitfalls, and these have to be understood in the context of the Observer pattern. Several questions arise, such as *How do you handle*

simultaneous notifications on multiple threads? What about modifying the listener list from one thread while notifications are in progress on another? and *What happens when the notification is sent from one thread to an object that is being used by a second thread?* These and other issues are discussed in [**?**].

10.8.4 Recognizing and processing external events

The entire process for receiving and processing input involves the following steps: *(i) providing a mechanism for input on the UI; (ii) listening to user actions on the input mechanism; (iii) generating appropriate internal events; (iv) processing the events.*

In our implementation, steps (i) through (iii) are performed in the UI and (iv) in the back-end. It is tempting to carry out (ii) through (iv) in the back-end, since it appears to make the process more efficient. However, from the point of view of reuse, that approach makes for a poor system and efficiency issues can be handled in other ways. Generally speaking, the UI is responsible for the "look and feel" and the back-end handles the processing. Reuse is most benefited if the back-end is "UI-agnostic," and that would be impossible if step (ii) is to be done in the back-end. A UI may provide a user with multiple mechanisms for the same operation (using a menu, a key sequence etc.) and the back-end should be able to handle all these uniformly. It is therefore desirable that steps (i) to (iii) be completed in the UI. The secondary question then arises as to which object in the UI should implement `actionListener`. This is addressed in one of the exercises.

Next we deal with the issue of communicating the event to the back-end. At first glance, the Observer pattern seems suitable for this, but a closer examination tells us that this may lead to a situation where the observer must use a conditional to distinguish between several observables. It is therefore preferable to use one of other mechanisms discussed in the chapter.

Projects

1. **Creating a controller for a Digital Camera.** A digital camera has several possible modes of operation. Each mode has its own interface, and the user can switch between modes. One mode, for instance, is the *Viewing* mode, in which stored pictures can be viewed, deleted, etc. In the *Setting* mode other parameters such as the kind of pictures to be taken, can be modified.

   - Study models of digital cameras on the market, and define a set of requirements for the UI.
   - Design a software controller that meets these specifications.

   Note that in such a system, both the view and the behavior will change when the state changes. Also, in a typical camera, the control buttons remain the same regardless of the mode of operation, but the effect of activating them changes. How will you model such functionality?

2. **A user interface for a Warehouse Management System.** In Chapter 6, a case-study for a warehouse database was presented. Create a complete GUI for such a system.

   - The UI has an initial login panel that allows the user to log in. The user could be a *client,* a *salesclerk* or a *manager.* Each type of user has a different set of access privileges. This will involve having some kind of password protection and can be accomplished using a separate subsystem that tracks the registered users and their passwords. The GUI will directly communicate with this system.
   - When a user logs in the appropriate menu is revealed. A salesclerk, for instance, performs operations like processing purchase orders from clients, receiving shipments etc. However, a salesclerk can become a particular client and do those operations too. The salesclerk menu should provide an option like "become a client". Likewise, a manager can become a salesclerk.
   - The GUI should have a "back" button to go back to a previous state.
   - Each menu panel should have a "logout" option.

   When a salesclerk becomes a client, this will require that we go through a panel that collects the particular client's ID. When the logout option is chosen, the state should go back to the salesclerk menu, whereas if the user was a client, the user will be logged out. Can this state be shared by the client and the salesclerk? How will you accomplish this? If the final choice of next state depends on stored information, where should this information be stored and in which class should the next state be computed? How is this impacted by the manner in which we are implementing the State pattern?

3. Implement a simple CD player. The player has the following buttons:

   (a) Insert/Eject: If a CD is inside the player, pressing this button causes the CD to be stopped and ejected. Otherwise, a CD is inserted and played.

   (b) Play: causes the player to resume playing a CD (if a CD is inside) from the position it was paused or from the beginning. If there is no CD, pressing this button has no effect.

   (c) Stop: causes the player to pause playing/fast-forwarding/rewinding, so pressing the Play button later causes the player to resume from this position. If this button is pressed when the player is paused, the CD is stopped, so a further push of the Play button plays the CD from the start. If there is no CD, pressing this button has no effect.

   (d) Fast Forward: If a CD is inside, the player plays the CD forward at double speed. Pressing this button while fast forwarding causes the player to resume playing again. If there is no CD, pressing this button has no effect.

   (e) Rewind: If a CD is inside, the player plays the CD backward at double speed. Pressing this button while rewinding causes the player to resume playing again. If there is no CD, pressing this button has no effect.

   All CDs play for exactly one hour. When the player reaches the end of the CD while playing or fast-forwarding, it stops (so it reverts to the start).

The user interface must be a GUI with the above five buttons and two displays: one showing the number of minutes and seconds elapsed if playing a CD and the other showing the state: like "playing," "paused," etc.

4. A room has the following options for climate-control: blow a fan, use an air-conditioner, employ a heater, or do nothing. A temperature regulator for the room operates can be set in one of four different modes to choose the desired option. (Imagine a slider control that can be set to one of the four positions.)

   (a) Do nothing: None of the three devices (fan, air-conditioner, and heater) is active.

   (b) Fan: The fan blows for ten minutes and then stays inactive for another ten 10 minutes; the cycle repeats.

   (c) Air-Conditioner: The air-conditioner immediately turn on. If the room temperature is too high, it operates an air conditioner until the room temperature hits the set temperature.

   (d) Heater: The heater immediately turn on. If the room is too cold, it operates a heater until the room temperature hits the set temperature.

   Apart from the four manual controls, assume that the regulator gets three other signals: room is too hot, room is too cold, and the temperature is just right.

   Develop the state transition table and diagram. Implement the system.

5. Implement a GUI for the library system.

Exercises

1. Modify the microwave implementation so that each button is an `actionListener` and performs the necessary actions when the button is clicked. How does this impact the overall complexity of the system?

2. Modify the implementation of the microwave controller so that individual states register with event sources. What changes will have to be made to the state classes? How will the states access the object with which they have to register/de-register themselves?

3. In the implementation of the State pattern for the microwave, the context keeps track of the current state, but the next state is decided by the current state. Suggest at least two other implementations of the State Pattern for the microwave. Compare and contrast all three implementations in the context of performance, simplicity of design and ease of reuse.

4. Modify the design of the microwave system to add each of the following requirements:

   (a) An "extend cooking" button is added to the display; if this button is pressed when cooking is in progress, 30 seconds are added to the cooking time.

(b) The system has a “clear” button, that sets the remaining cooking time to zero. The system also stores the remaining time if the cooking is interrupted by the opening of the door. When the system enters the cooking state again, this stored value is used as the cooking time; however, if the clear button has been pressed in the meantime, it runs for 60 seconds.

(c) The system displays an “error” message if an inappropriate action is performed. For instance, if the cook button is pressed when the door is open, the message “Please close the door” is displayed.