Chapter 10
Modelling 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 two case-studies of systems where the use case model does not suffice. The first of these is 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 microwave does not always fire up. 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 Graphical User Interfaces (GUI), which are used by applications to interact with a user. 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 analysing such systems.

10.2 A Simple Example

The use of software to control the behaviour of systems is well known. Such systems can comprise several hardware components, each of which may be turned on or off using embedded software. The system as a whole has to behave in a prescribed

B. Dathan and S. Ramnath, *Object-Oriented Analysis, Design and Implementation*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-24280-4_10

manner, turning components on or off depending on the input and the other environmental variables. The following is a simple example of such a system.

Problem Consider a simple microwave oven whose behaviour 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.
- 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 power-tube is turned off.
- 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 s → open door → remove food and stir → place food in oven → close door → push button → wait for 45 s → open door → remove food → close door → stir food → open door → place food in oven → push button → wait for cooking to finish → open door → remove food → close door

Clearly, there is no set of standard 'business processes' that can characterise 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 system is. The system may also change state in response to these events. What this suggests is that in order to model the system behaviour accurately, we should treat this as a **finite state machine (FSM)**.

10.3 Finite State Modelling

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. We shall use a Mealy machine for modelling our FSM.

Returning to our microwave oven, we now try to identify the states in the FSM that would model the behaviour of the microwave oven. An initial examination yields the following possible states:

1. Microwave is idle and the door is closed.
2. Microwave is idle and the door is open.
3. Microwave is in operation.
4. Microwave is interrupted by the door being opened.
5. 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:

- door is opened
- door is closed
- button is pushed
- clock ticks
- timer runs out

The first three are external events that are 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 Fig. 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 external events. 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 Fig. 10.1 is given using the UML state transition diagram in Fig. 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.

	Open door	Close door	Press cook	Clock 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	Idle; Door closed
Interrupted	Idle; Door open	Idle; Door closed	Idle; Door open	Idle; Door open	Idle; Door open
Completed	Idle; Door open	Idle; Door closed	Cooking	Idle; Door closed	Idle; Door closed

Fig. 10.1 Transition table for the microwave

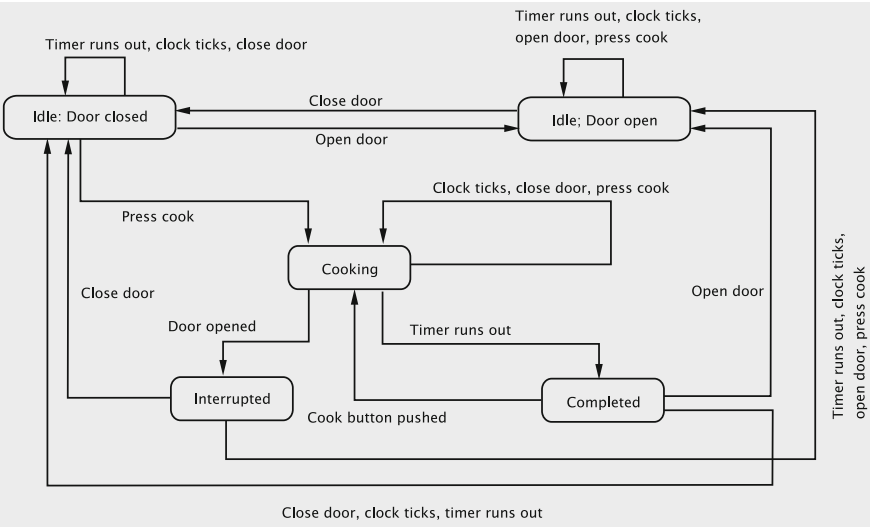

Fig. 10.2 State transition diagram for the microwave oven

One observation we make here is that the behaviour 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 behaviour of the system when it is idle with door closed from the behaviour 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 Fig. 10.3.

	Open door	Close door	Press 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

Fig. 10.3 Minimised transition table for the microwave oven

In general, it is important to find an FSM with a small number of states and there are exact algorithms for state minimisation.[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.

10.4 A First Solution to the Microwave Problem

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 Chaps. 6 and 7, we start by constructing the list of nouns: Microwave, Powertube, Light, Display, Door, CookButton, etc. The Display and CookButton will be part of the user interface (GUI). Since this is only a 'software simulation', we cannot have a real Powertube or Light, and so we simply have to model these by displaying some message on the GUI. (To make it more realistic,

Use-Case Modelling Versus Finite State Modelling
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` object, 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

[1] The reader is referred to any text on *digital logic design* or *automata theory.*

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 behavioural response). Also, we typically have only a 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 modelling is inappropriate and that using an FSM to model the system behaviour would be the best choice.

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.

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 a clock tick event 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 themselves seem to serve our purpose well. The next step is to figure out how the software classes will distribute the responsibilities to achieve the behaviour 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 recognised by the GUI.
- *Clock Ticks*: these originate in `Clock`.

To describe the manner in which the entities of the system handle these, we shall 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. When the display is updated, we shall assume that the actor can see the result of the action.

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 give us enough information to specify the responsibilities of individual classes. `Microwave` is a singleton class with methods to process the external 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 Fig. 10.6.

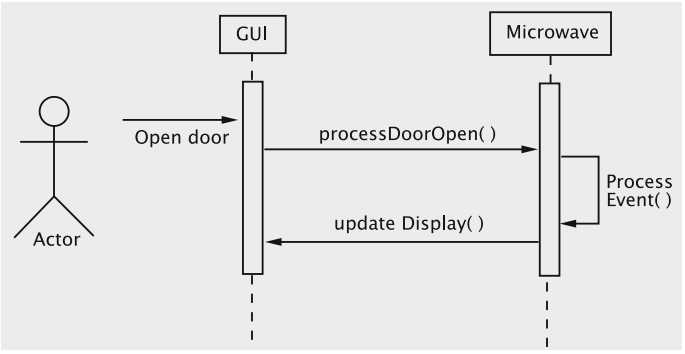

Fig. 10.4 Sequence diagram for door opening

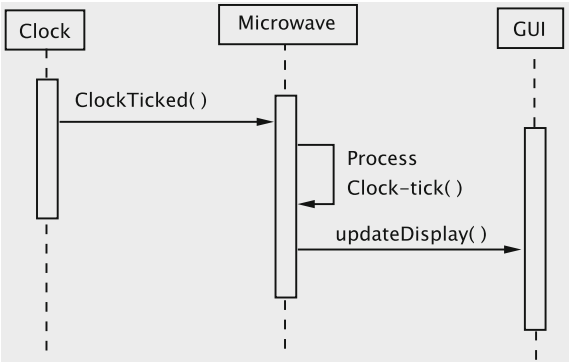

Fig. 10.5 Sequence diagram for processing a clock tick

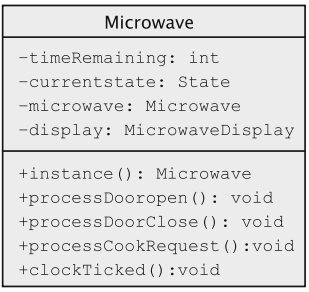

Fig. 10.6 Microwave class diagram

Although a text-based interface is impractical, 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 sequence diagram simply shows a method `updateDisplay`, but a little thought would show that it is better to have a set of methods to independently set the several displays such as the status of the light, powertube, etc.

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 UI 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 Fig. 10.7. The method `setMicrowave` configures the display with an instance of `Microwave`. All the other methods are provided to display to the user the current status of the system.

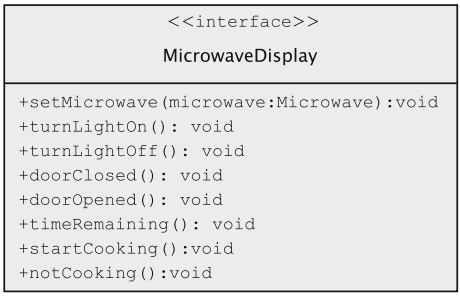

Fig. 10.7 Microwave display interface

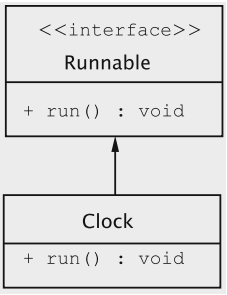

Fig. 10.8 Clock

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 Fig. 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. In its constructor, the `Clock` object gets hold of the reference to the `Microwave`, which is a singleton. The `run` method is an infinite loop waking up every second and invokes the `clockTicked` method on `Microwave`. The code is given below.

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

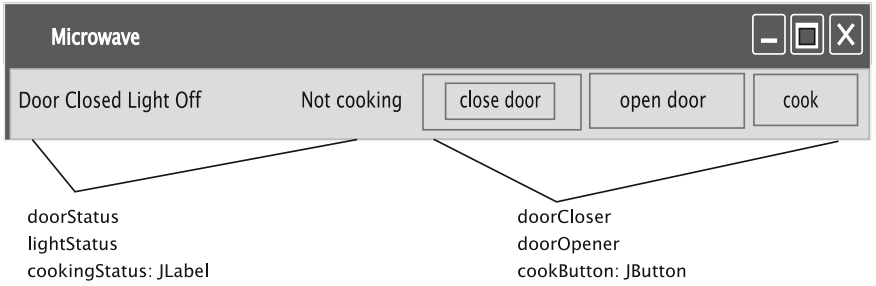

Fig. 10.9 Microwave interface

```
      } catch(InterruptedException ie) {
      }
    }
  }
```

The Display Class

`GUIDisplay` is the concrete class that implements `MicrowaveDisplay`. To handle user input, it creates a `JFrame` with a `JButton` for each kind of operation: open door, close door, and cook.

When run, the program displays the interface given in Fig. 10.9. It has `JLabel` fields for displaying the status.

```
public class GUIDisplay extends JFrame
    implements ActionListener, MicrowaveDisplay {
  private Microwave microwave;
  private JButton doorCloser = new JButton("close door");
  private JButton doorOpener = new JButton("open door");
  private JButton cookButton = new JButton("cook");
  private JLabel doorStatus = new JLabel("Door Closed");
  private JLabel timerValue = new JLabel("            ");
  private JLabel lightStatus = new JLabel("Light Off");
  private JLabel cookingStatus = new JLabel("Not cooking");
  // other fields and methods
}
```

The constructor lays out all the widgets and sets the `GUIDisplay` object to be the `ActionListener` for all the `JButton` objects.

```
public GUIDisplay() {
  super("Microwave");
  addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent event) {
      System.exit(0);
    }
  });
  getContentPane().setLayout(new FlowLayout());
  getContentPane().add(doorStatus);
  getContentPane().add(lightStatus);
  getContentPane().add(timerValue);
  getContentPane().add(cookingStatus);
  getContentPane().add(doorCloser);
  getContentPane().add(doorOpener);
```

```
    getContentPane().add(cookButton);
    doorCloser.addActionListener(this);
    doorOpener.addActionListener(this);
    cookButton.addActionListener(this);
    pack();
    setVisible(true);
  }
```

The `actionPerformed` method checks the source of the `ActionEvent` and invokes the appropriate method of `Microwave`.

```
  public void actionPerformed(ActionEvent event) {
    if (event.getSource().equals(doorCloser)) {
      microwave.processDoorClose();
    } else if (event.getSource().equals(doorOpener)) {
      microwave.processDoorOpen();
    } else if (event.getSource().equals(cookButton)) {
      microwave.processCookRequest();
    }
  }
```

It is tempting to make `Microwave` the listener for the button clicks, and skip the above step. Note that this would make `Microwave` tightly coupled to the `GUIDisplay` and is therefore undesirable. Finally, we need methods to update the values in the `JLabel` objects that display the system status. This is accomplished by implementing the methods in the `MicrowaveDisplay` interface. As an example, the code for `turnLightOn` is shown below.

```
  public void turnLightOn() {
    lightStatus.setText("Light On");
  }
```

The Microwave Class

Now we discuss the more involved class, `Microwave`. The class maintains the variables for keeping track of the remaining cooking time and the current state. A concrete class that implements `MicrowaveDisplay`, viz. `GUIDisplay`, is instantiated and a reference to it is stored.

```
  public enum States {DOOR_CLOSED_STATE, DOOR_OPENED_STATE, COOKING_STATE};
  private int timeRemaining;
  private States currentState;
  private static Microwave instance;
  private MicrowaveDisplay display;

  private Microwave() {
    currentState = States.DOOR_CLOSED_STATE;
    timeRemaining = 0;
    display = new GUIDisplay();
    display.setMicrowave(this);
    display.timeRemaining(timeRemaining);
    display.turnLightOff(); display.doorClosed();
    display.notCooking();
  }
```

```
public static Microwave instance() {
  if (instance == null) {
    return instance = new Microwave();
  }
  return instance;
}
```

Next we look at the code for processing each of the events. When the clock ticks, `Microwave` needs to take action only if it is in the `COOKING_STATE`; this action involves decrementing the remaining time, updating the display, and if the timer has run out, switching to the `DOOR_CLOSED_STATE`. In case of this switch, the display needs to be updated once again. This code is shown below.

```
public void clockTicked(){
  if (currentState == States.COOKING_STATE){
    timeRemaining--;
    display.timeRemaining(timeRemaining);
    if (timeRemaining == 0) {
      currentState = States.DOOR_CLOSED_STATE;
      display.notCooking();
      display.turnLightOff();
    }
  }
}
```

The methods for processing the other events follow a similar pattern. In each case, `Microwave` checks the current state and takes the appropriate action. If the action results in a change of state, some transitional work also needs to be done. As a second example, consider the `processCookRequest` method. The cooking request is processed only if the system is in the `COOKING_STATE` or the `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 s are added to the time remaining and the display is updated.

```
public void processCookRequest() {
  if (currentState == States.DOOR_CLOSED_STATE) {
    currentState = States.COOKING_STATE;
    display.startCooking();
    display.turnLightOn();
    timeRemaining = 60;
    display.timeRemaining(timeRemaining);
  } else if (currentState == States.COOKING_STATE) {
    timeRemaining += 60;
    display.timeRemaining(timeRemaining);
  }
}
```

The methods for processing the opening and closing of the door are similar and we leave those as an exercise. The `Microwave` class also has the `main` method that gets the show going; this is done by instantiating `Clock`.

```
public static void main(String[] args) {
  new Clock();
}
```

10.4.4 A Critique of the Above Design

The above solution is our first attempt at solving this problem. We have correctly analysed 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 recognising and applying appropriate design patterns.

Extreme 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 y 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 behaviour, and succeeded in reducing the complexity of the individual methods and also in facilitating reuse. The question that arises therefore is: *Can we avoid the conditionals that switch on* `currentState` *by subclassing?* The answer is not quite obvious here, since we are dealing with a *dynamic* situation, i.e., the value of `currentState` changes with time, unlike the field `bookType` in the previous chapter which was finalised in the constructor. As it turns out, such a subclassing is indeed possible, but some additional machinery is needed to manage the changes in the value of `currentState`. All of this is handled by using the **state pattern**.

Communication Between 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 Close Door button.
2. Events of a more general nature that could find relevance in any application. The only such event 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. The bigger concern is with the fact that this hurts *reuse.* 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 several applications. In general, designs like the one we have 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. Instead, we should 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 Using the State Pattern

Using the state pattern is closely connected to the idea of modelling with FSMs. All these situations therefore have the following characteristic elements:

1. A collection of states, with each state being defined by distinct behaviour.
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 addition, the context also serves as a facade for the entire 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 attributes:

1. A field to track the current state of the FSM.
2. Provide a mechanism to record the change of state.

These two attributes are essential. In addition, the context may provide other attributes 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.

10.5.1 Creating the State Hierarchy

In the solution given in Sect. 10.4, the `Microwave` class had a variable `currentState` on which the behaviour was conditionally executed. This is reminiscent of the use of `bookType` in the library system in Chap. 9. 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 Fig. 10.10.
2. A second part, which deals with the contextual information. We could view the original `Microwave` class now as simply holding the contextual information required for the operation of the FSM. It is therefore aptly renamed `MicrowaveContext`.

Transitioning between states Transitioning to a new state is an operation that involves the knowledge of the other states. Before we decide how to implement

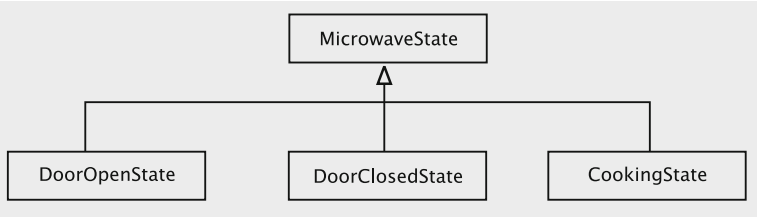

Fig. 10.10 MicrowaveState hierarchy

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 correspond roughly to the two standard methods for storing directed graphs: **adjacency matrices** and **adjacency lists**. In an adjacency matrix, we have a centralised 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 neighbours. These two representations lead to two possible implementations for handling transitions between states.

Using the matrix representation

In this approach we first associate an index with each state. It is a common convention to use the index 0 for the initial state, so we assign the index 0 to the closed door state, index 1 to the open door state and 2 to the cooking state. To keep track of this mapping, we create an array; thus the context has an array of `MicrowaveState` named `state` such that `state[0]` would store a reference to the Door Closed state, `state[1]` would store a reference to the Door Open state, and so on. 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 (Fig. 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}};
```

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 `changeCurrentState` method on the context, passing the index of the needed transition. The code for `changeCurrentState` would be something like this:

```
public void changeCurrentState(int next) {
  currentState = transitions[currentState][next];
  state[currentState].run();
}
```

The parameter `next` 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 `state` and the `run` method is invoked on the new state.

Note that we are dealing with both *external* and *internal* events. The events Open Door, Close Door, Press Cook and Clock Ticks are external to the FSM, and correspond to real-world events that occur in other subsystems, viz., the GUI and the Clock. The event Timer Runs Out occurs when `timeRemaining` drops to zero; since `timeRemaining` is tracked only inside the FSM, this event is detected internally. This does not pose any difficulty as far as our implementation is concerned. All that is needed is that the state that detects the internal event (Cooking state, in our case) generates the appropriate index (4, in our case) and uses this index as the argument when invoking `changeCurrentState`.

Using the List Representation

In this representation, each state directly provides a reference to the next state when it has to relinquish control. Each concrete state is implemented as a singleton, and therefore the `instance` method can be used for that purpose. The context, of course needs to be informed of the change of state, and this is done once again using the `changeCurrentState` method.

```
public void changeCurrentState(MicrowaveState state) {
  currentState = state;
  currentState.run();
}
```

The context stores a reference to the current state and invokes the `run` method as before.

Both approaches have been referred to in the literature and have their pros and cons. In the matrix approach, each state can be written independently of the others. The `run` method sets up the state with the help of the context, and we have methods for processing all the necessary events. This coding occurs in the driver routine, and is used to populate the transition table. This has the advantage that the code for a state does not have to be modified unless we want to change its behaviour. The transitions can be changed and new states can be linked to existing ones in the driver routine itself. 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 list 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 model something specific like, say, an algorithm for a 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 list approach and embed the transition information within each state. *We shall implement the transitions for our case-study using the list approach.*

State classes We now elaborate on the state classes. The abstract class, `MicrowaveState`, contains methods to handle the various events; its class diagram is shown in Fig. 10.11. The meanings of most of the methods should be obvious. When the microwave changes state, some variables may need to be initialised and the output may have to be changed. It is convenient to have all of these actions executed in a method, which we term `run`.

Next, we develop the class diagrams for the individual states. While in the DoorOpen state, the microwave does not respond to anything other than the door closing. Therefore, in the class diagram for `DoorOpenState` (Fig. 10.12), we show the methods `processDoorClose` and `run`. Similar interpretations can be made for `DoorClosedState` and `CookingState`, which are respectively shown in Figs. 10.13 and 10.14.

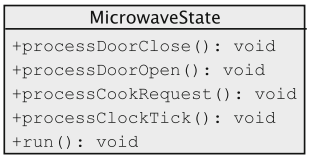

Fig. 10.11 Class diagram for MicrowaveState

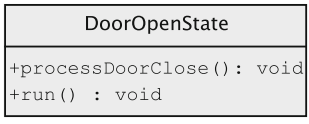

Fig. 10.12 Class diagram for DoorOpenState

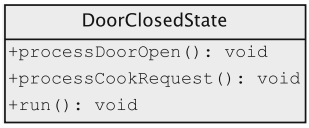

Fig. 10.13 Class diagram for DoorClosedState

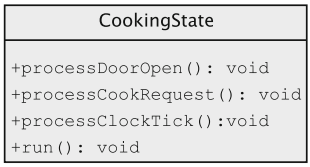

Fig. 10.14 Class diagram for CookingState

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.*

Changing the state is handled using one of the mechanisms described previously. (We chose the adjacency list approach.)

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 second 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.*

If we choose the first option, we have a context that truly behaves like a *facade*, i.e., all communication into and out of the system goes through the context. This 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 powertube 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, but such a call can be easily *inlined* to reduce the runtime cost.

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, making 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 above discussion leads to a natural question: *Can incoming communication go directly to the current state?* This is clearly a tricky question, since the external entity does not know what the current state is, and revealing such information would clearly lead to a lot of unwanted coupling. As it turns out, these questions are related and we shall examine all of this in Sect. 10.6 when we deal with the issue of communication. For now, we shall make the choice that *all communication goes through the context.* What this implies is that the context has methods for updating the display, which are invoked from the concrete states.

The class diagram for `MicrowaveContext` is shown in Fig. 10.15. 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 `getDisplay` to provide the reference to the display object, which can be used by the the `MicrowaveState` objects as needed to update the interface. The current state can be changed by executing the method `changeCurrentState`.

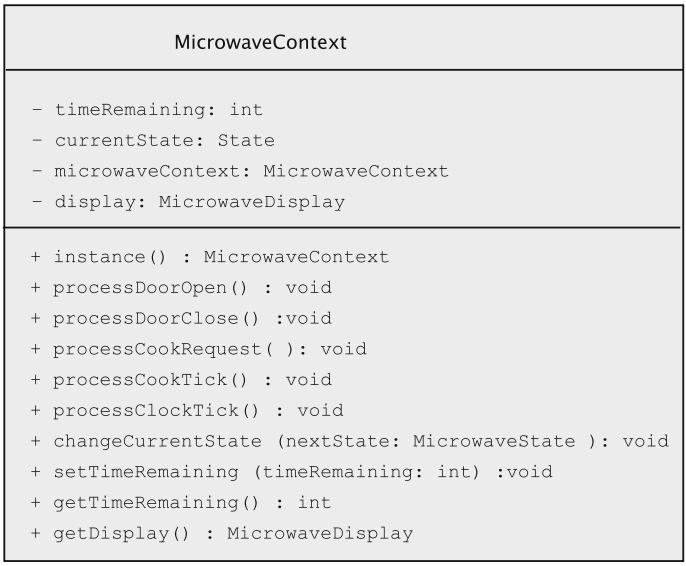

Fig. 10.15 Class diagram for MicrowaveContext

10.5.2 Implementation

`MicrowaveState` has default methods for processing each of the events, and the concrete state classes are supposed to override these to specify the processing required.

```
public abstract class MicrowaveState {
  protected static MicrowaveContext context;
  protected static MicrowaveDisplay display;
  protected MicrowaveState() {
    context = MicrowaveContext.instance();
    display = context.getDisplay();
  }
  public abstract void run();
  public  void processDoorClose() {
  }
  public  void processDoorOpen() {
  }
  public  void processCookRequest() {
  }
  public  void processClockTick() {
  }
}
```

The `run` method is abstract and is invoked on a state whenever control has to be transferred to that state. Each state must therefore define the `run` method in an appropriate manner. For instance, when the cooking state is entered, the housekeeping needed is that the light must be turned on, the timer set to 60, and powertube turned on. These details are shown below.

```
public  void run() {
  display.turnLightOn();
  context.setTimeRemaining(60);
  display.startCooking();
  display.displayTimeRemaining(context.getTimeRemaining());
}
```

The default methods are overridden as needed. The method `processClockTick` in `CookingState`, for instance, would look something like this:

```
public  void processClockTick() {
  context.setTimeRemaining(context.getTimeRemaining() - 1);
  display.displayTimeRemaining(context.getTimeRemaining());
  if (context.getTimeRemaining() == 0) {
    display.notCooking();
    display.turnLightOff();
    context.changeCurrentState(DoorClosedState.instance());
  }
}
```

The `MicrowaveContext` class holds a reference to the current state and processes the events by simply passing on the request to the current state. The code for the method `processCookRequest` now looks like this:

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

The variable `currentState` is of type `MicrowaveState` and dynamic binding ensures that correct method is invoked. Note that we no longer have a conditional to check what kind of state we are in. Note that some states (e.g., Door Open) are required to ignore this event; for this reason the default method in the abstract class is implemented to do nothing.

Implementing the concrete states Each of these extends `MicrowaveState` and is a singleton. Here are the first two lines of the code for the `CookingState` class.

```
public class CookingState extends MicrowaveState {
  private static CookingState instance;
```

Each state class overrides only a subset of the methods of `MicrowaveState`. The code for handling the external events is essentially the same as what we had in the corresponding methods of `Microwave` in the previous version: the conditional on `currentState` is now absent. Clock ticks are handled in `CookingState` as shown below:

```
public void processClockTick() {
  context.setTimeRemaining(context.getTimeRemaining() - 1);
  display.displayTimeRemaining(context.getTimeRemaining());
  if (context.getTimeRemaining() == 0) {
    context.changeCurrentState(DoorClosedState.instance());
  }
}
```

In this version, we shall leave the `Clock` class largely unchanged. The minor, obvious change is that rather than send signals to the `Microwave` object, `Clock` has to notify the instance of `MicrowaveContext`.

10.6 Improving Communication Between Objects

As we discussed earlier, we have two problems with communication in our earlier design. The first of these involves external entities like `Clock`. In our example, this was tightly coupled to the implementation for `Microwave`, and we shall investigate loosening this coupling using the *Observer* pattern. Next we look at how our system can be made more flexible by taking an *event-based approach* to processing user input.

10.6.1 Loosely Coupled Communication

Loosely coupled communication must have three properties:

1. The listener is responsible for registering interest.
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. 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 this interest has been registered, it is the responsibility of the observable to notify all observers of any changes/events that occur. In order to accomplish this without causing tight coupling, every observer must have a method with a signature that has been agreed upon. Java provides such a mechanism as a part of the language.

Every `Observable` object maintains a list of 'interested observers.' The class `Observable` has the following categories of methods:

1. In the first category, we have methods for maintaining the list of observers. The method `addObserver(Observer observer)` adds the given observer to this list. There are two ways of deleting observers. A single observer can be deleted by using the method `deleteObserver(Observer observer)`, and all of the observers can be deleted by calling the method `deleteObservers()`.
2. The second set of methods support the notification of the observers of 'noteworthy' events occurring within the `Observable` object. Every `Observable` object maintains a flag, 'object changed', to remember whether the object has changed since the last notification to the observers. To notify observers, it is necessary to first call the method `setChanged()` to indicate that a change has occurred. This sets the 'object changed' flag to `true` . After setting the flag, the `Observable` can notify all of the observers in one of two ways: by calling `notifyObservers()` or calling `notifyObservers(Object arg)` . In the latter version of the `notifyObservers` method, `arg` is the object that contains the message to be delivered. Every time `notifyObservers` is invoked, the 'object changed' flag is cleared; that is why we need to call the `setChanged` method whenever a 'noteworthy' change/event occurs. (Repeated invocations of the `notifyObservers` method will not have any effect until the 'object changed' flag is set again.)
3. The third group comprises miscellaneous methods such as `countObservers()` that gets the number of observers of the `Observable` object.

Every class that wishes to be an observer implements the `Observer` interface. The `Observer` interface has the method `update` with the following signature:

```
public abstract void update(Observable object, Object arg);
```

When `notifyObservers` is invoked in the `Observable` object, the `update` method is invoked once for each item in the list of 'interested observers'. The `update` method allows the `Observable` object to send a reference to itself along with a message to the observer.

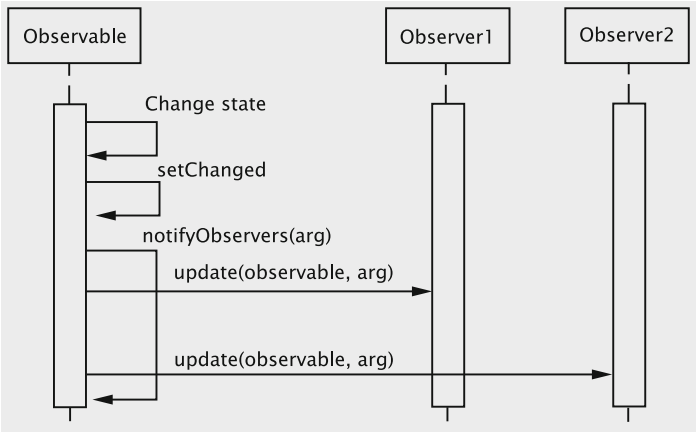

Fig. 10.16 Sequence diagram in a system with one observable and two observers

The interaction between and `Observable` and two `Observer` objects is depicted in the sequence diagram in Fig. 10.16. The picture shows an event occurring in the `Observable` , which calls the `setChanged` and the `notifyObservers` methods. In response to the `notifyObservers` method call, the `Observable` object calls the `update` methods of the two observers supplying them with its identity and information about the event(`arg`).

While utilising the observer pattern in Java, the programmer should be aware of two aspects.

1. In general, a single observer should not be registered more than once with the same `Observable` object. The `Observable` object will call the `update` method of the observer as many times as the number of registrations. The problem could creep into applications where observers are supposed to be deleted and then put back in.
2. The `update` method of the observer is executed by the `Observable` thread. This means that the processing of the `update` method in the observers occurs sequentially. If one of these methods ends up waiting, say for a message, the `Observable` object will be stuck! Also, the designer must ensure that no matter in what order the `update` methods get executed, the system will function correctly.

10.7 Redesign Using the Observer Pattern

We now re-implement the microwave system to have the `Clock` extend `Observable` and `MicrowaveContext` implement `Observer`. Since the `Clock` object does not maintain a reference to the context anymore, `Microwave`

Context need not be a singleton. On the other hand, Clock is implemented as a singleton because it can serve as a general class for a variety of applications. This allows the listener (observer) to get hold of the event generator (observable) and register its interest. The modified Clock class is as follows:

```
public class Clock extends Observable implements Runnable {
  private Thread thread = new Thread(this);
  private static Clock instance;
  public enum Events {CLOCK_TICKED_EVENT};
  private Clock(){
    thread.start();
  }
  public static Clock instance() {
    if (instance == null) {
      instance = new Clock();
    }
    return instance;
  }
  public void run(){
    try{
      while (true) {
        Thread.sleep(1000);
        setChanged();
        notifyObservers(Events.CLOCK_TICKED_EVENT);
      }
    } catch (InterruptedException ie) {
    }
  }
}
```

MicrowaveContext implements the Observer interface and the method ClockTicked is replaced by the method update.

```
public class MicrowaveContext implements Observer {
  public void  update(Observable source, Object event) {
    // code to process clock tick
  }
  // other attributes as before
}
```

10.7.1 Communication with the User

As we discussed earlier, implementing the communication mechanism requires us to make a decision about the path of the messages. We had decided that all communication would go through the context. We shall now proceed with that assumption and try to improve the reusability of our system.

As it stands, we have provided a separate method in MicrowaveContext for each kind of event. We have a situation where adding new kinds of events would require changing the implementation of the context. The simplest way of avoiding this is to have a single method, say processEvent, and pass the event as a parameter.

This means that all the events should belong to a common type, which can be created using the `enum Event`, as shown below:

```
public class MicrowaveSupport {
  public static enum Events {DOOR_CLOSED_EVENT, DOOR_OPENED_EVENT,
                             COOKING_REQUESTED_EVENT};
}
```

The class `MicrowaveSupport` is defined to hold all the events. Note that what we have done is to *encapsulate the variability in a separate class* so that new events can be added without changing the context. The `processEvent` method in the context simply passes the event on to the current state.

```
public void handleEvent(Object arg) {
  currentState.handle(arg);
}
```

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.

Note that the context does not distinguish between the events anymore. Accordingly, the abstract class `MicrowaveState` has just a single abstract method `handle` which must be overridden by each concrete state. The code for the class `CookingState` is shown below. (It assumes that `MicrowaveContext` is still a singleton.)

```
public void handle(Object event) {
  if (event.equals(MicrowaveSupport.Events.COOKING_REQUESTED_EVENT)) {
    processCookRequest();
  } else if (event.equals(MicrowaveSupport.Events.DOOR_OPENED_EVENT)) {
    processDoorOpen();
  } else if (event.equals(Clock.Events.CLOCK_TICKED_EVENT)) {
    processClockTick();
  }
}
```

Note that we now have conditionals to distinguish between the events. This is because all the events are travelling along the same path and we have no means of distinguishing them. A similar situation occurs when we have a class that acts as an observer for several observable classes. The advantage to using the `enum` is the simplicity of implementation when compared with the other options; the disadvantage is that we combine the complexity of several methods into one, requiring switching on the type of event. This complexity is not a big concern when we have a simple system and each state handles only a small number of events, but can be a problem when there are several events to be handled and these originate from divers entities. As one would expect, we have alternate solutions for this that involve creating a separate event hierarchy. We shall examine these options later in this chapter.

10.7.2 The Improved Design

Before proceeding further, let us summarise the changes we have made to the earlier design by examining each class.

The GUIDisplay This class has remained largely unchanged. The only difference is that instead of invoking different methods for processing each event, we now invoke the same method `processEvent` with different parameters. This code is shown below:

```
public void actionPerformed(ActionEvent event) {
  if (event.getSource().equals(frame.doorCloser)) {
    MicrowaveContext.instance().handleEvent(
          MicrowaveSupport.Events.DOOR_CLOSED_EVENT);
  } else if (event.getSource().equals(frame.doorOpener)) {
    MicrowaveContext.instance().handleEvent(
          MicrowaveSupport.Events.DOOR_OPENED_EVENT);
  } else if (event.getSource().equals(frame.cookButton)) {
    MicrowaveContext.instance().handleEvent(
          MicrowaveSupport.Events.COOKING_REQUESTED_EVENT);
  }
}
```

Clock This class now extends `Observable` and is not tailored for this particular application, i.e., when a clock tick happens, it simply notifies all the observers.

MicrowaveContext The class was set up as an `Observer` of `Clock`. The variability in the events was encapsulated, and a single method, `processEvent`, was defined in `MicrowaveContext` to receive all events. A separate hierarchy, viz., `MicrowaveState`, was created and all the details of processing of events in the different states was delegated to this hierarchy. A class diagram showing all these details is presented in Fig. 10.17.

The manner in which events are handled changes in conformance with the new implementation. The sequence diagram for handling a cooking request event when the microwave is in the `DoorClosedState` is shown in Fig. 10.18.

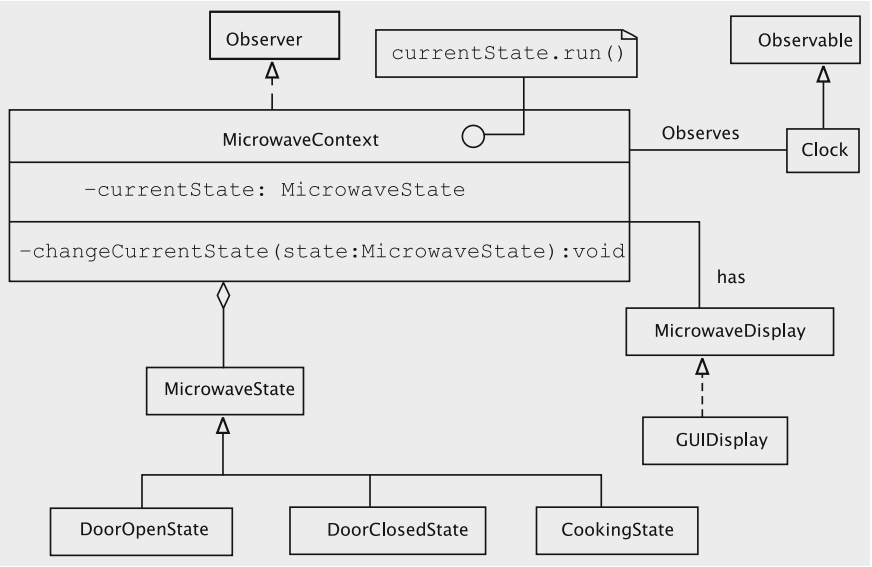

Fig. 10.17 Class diagram for the microwave oven

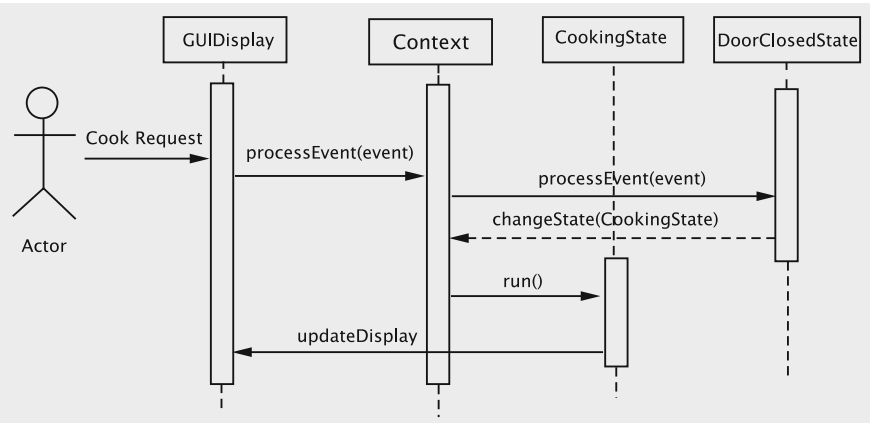

Fig. 10.18 Sequence diagram for cooking request event

10.8 Eliminating the Conditionals

In the design presented above, the `handle` method in each state handles the events. This conditional switches on the kind of event, which in turn is decided by external input. To eliminate these conditionals from the states, it is obvious that our implementation should ensure that each event is delivered to the states in a unique way. Since the events are passed on through method invocations, this would mean having

a unique method invocation for each event. However, the context cannot distinguish between these events for reasons discussed earlier, and this implies that we can no longer insist that all communication go through the context. The problem we are faced with is that of ensuring that only the current (i.e., active) state processes the event. We have two broad approaches for ensuring this.

1. *Take the context out of the picture completely i.e., each state takes full responsibility for performing exactly as the system requires.* This approach would use something like the event handling system provided by Java, which behaves very much like a customised observer pattern. We have an event source and an event listener interface which correspond respectively to the observable and the observer. The listener classes must implement the event listener interface and register themselves with the source object; when the event occurs, the method in the event listener interface is invoked on all the registered listeners. If we use this approach for our microwave oven, we will have a different event listener method for each event. For the reasons mentioned earlier, these cannot be implemented by the context. Each state therefore implements the needed event listener interfaces and also takes responsibility for registering and de-registering itself with the event sources.
2. *Use the context as a 'switchboard' that connects the event with the current state.* This would clearly have to be done without knowing the type of the concrete event or the concrete state, thus requiring some form of double dispatch. We create a hierarchy of events and a parallel hierarchy of listener interfaces. Each state implements the necessary interfaces as before. Since the concrete `MicrowaveState` object that is currently active changes constantly, we need to use a two-step process in which the source communicates the event to the context, which then invokes a method on the event object passing the current state as a parameter. The method in the event then invokes the method in the listener interface.

10.8.1 Using the Java Event Mechanism

In this approach, the current state object receives notifications directly from the source of the event, without going through the context. For each type of event, there is a separate 'manager' object that takes on the responsibility of delivering the events to the appropriate state. For example, consider a click on the door close button. The display object informs the `DoorCloseManager` object, which notifies all the listeners.

The Java event mechanism involves creating classes for the events and their sources and interfaces that the listeners will implement. Let us go through each of this, making the necessary modifications to our microwave system.

The Event Classes

We create a class for each event, by extending `EventObject`. The code for the `DoorCloseEvent` is shown below.

```
public class DoorCloseEvent extends EventObject {
  public DoorCloseEvent(Object source) {
    super(source);
  }
}
```

The only work here is to define the constructor. The source of the event is stored in the superclass object.

Concrete Versus 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 the default behaviour 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 helps with reuse.

In the observer pattern, the observable is an abstract class. The mechanism used to enforce the pattern requires that 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 behaviour, 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.

The Event Source
In our system, the events will be generated in response to button clicks. We can designate `GUIDisplay` as the source. It is customary that responsibilities of the source include providing mechanisms for registering and de-registering listeners and also notifying them when an event occurs. Doing this for all the microwave events in the GUI, however, causes unnecessary entanglement of responsibilities. Instead, we create a separate class `DoorCloseManager`, which handles all the other responsibilities. This class is created by extending `JComponent` so that we can inherit some of the functionality for managing the events.

```
public class DoorCloseManager extends JComponent {
  private EventListenerList listenerList;
```

```
  private static DoorCloseManager instance;
  private DoorCloseManager() {
    listenerList = new EventListenerList();
  }
  public static DoorCloseManager instance() {
    if (instance == null) {
      return instance = new DoorCloseManager();
    }
    return instance;
  }
  public void addDoorCloseListener(DoorCloseListener listener) {
    listenerList.add(DoorCloseListener.class, listener);
  }
  public void removeDoorCloseListener(DoorCloseListener listener) {
    listenerList.remove(DoorCloseListener.class, listener);
  }
  public void processEvent(DoorCloseEvent event) {
    EventListener[] listeners
        = listenerList.getListeners(DoorCloseListener.class);
    for (int index = 0; index < listeners.length; index++) {
      ((DoorCloseListener) listeners[i]).doorClosed(event);
    }
  }
}
```

This class is defined as a singleton, so that the concrete states can access it and register themselves. The event constructors are invoked when the button clicks are detected in the GUI, as shown below.

```
public void actionPerformed(ActionEvent event) {
  if (event.getSource().equals(doorCloser)) {
     DoorCloseManager.instance().processEvent(new DoorCloseEvent(this));
  }
  // code for handling other events
}
```

The `GUIDisplay` object is stored as the source, which enables us to track the actual source, if needed for an exception. The code above invokes the `processEvent` method of the `DoorCloseManager` which then notifies all the listeners.

The Event Listeners

All event listeners must implement the corresponding listener interface. All these interfaces are defined to extend `EventListener` so that the methods in `JComponent` can be reused.

```
public interface DoorCloseListener extends EventListener {
  public void doorClosed(DoorCloseEvent event);
}
```

Each concrete state implements the required listeners and does the necessary housekeeping. Note that this includes registering as a listener within the `run` method, and de-registering before the `changeCurrentState` method is invoked.

```
public class DoorOpenState extends MicrowaveState
                           implements DoorCloseListener {
  // other fields and methods
```

```
    public  void run() {
      display.stopCooking();
      display.openDoor();
      display.turnLightOn();
      display.displayTimeRemaining(context.getTimeRemaining());
      DoorCloseManager.instance().addDoorCloseListener(this);
    }
    public void doorClosed(DoorCloseEvent event){
      DoorCloseManager.instance().removeDoorCloseListener(this);
      context.changeCurrentState(DoorClosedState.instance());
    }
  }
```

10.8.2 Using the Context As a 'Switchboard'

Sometimes, we may have to deal with situations where the communication has to go through a facade. In such an FSM, we cannot use the Java event structure unless the context participates in the process and is aware of all kinds of events in the system, which may not be desirable. As it turns out, we can construct a solution where the context has a single `handleEvent` method and each individual state takes care of implementing the listener methods it is interested in. This can be accomplished by adding some machinery to the event classes and paying the price of an additional method call. Such a system consists of the following elements.

1. **An event hierarchy** We have an abstract class `MicrowaveEvent`, which is extended by the concrete event classes `DoorOpenEvent`, `DoorCloseEvent`, etc. This is shown in Fig. 10.19.
2. **A listener interface hierarchy** The hierarchy parallels the event hierarchy and is shown in Fig. 10.20. We have a general interface for `MicrowaveEvent Listener` which corresponds to the abstract event, extended by specific interfaces for each concrete event. We need the general interface, since the abstract class `MicrowaveState` has to be a listener but cannot implement any of the specific interfaces.

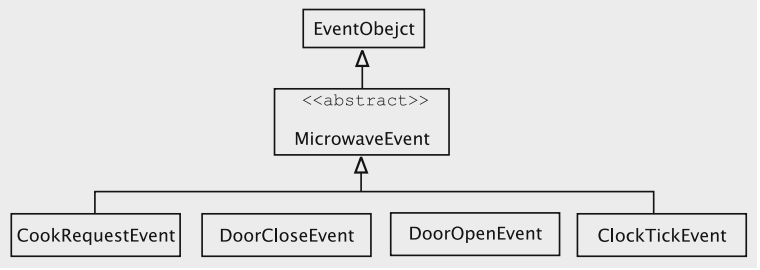

Fig. 10.19 Event hierarchy for the microwave oven

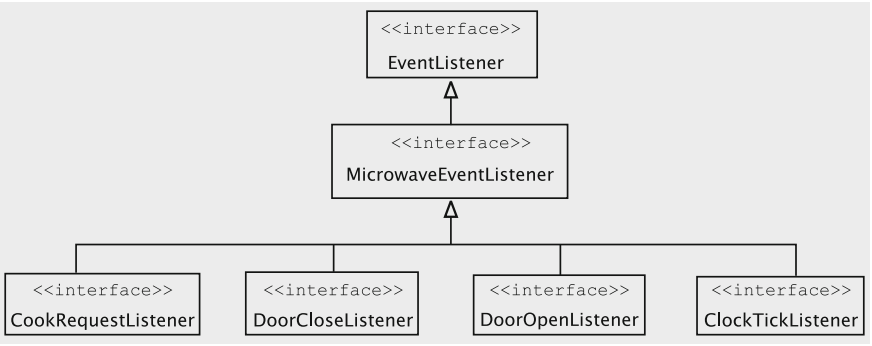

Fig. 10.20 Listener hierarchy for the microwave oven

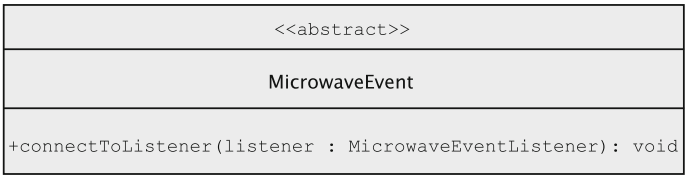

Fig. 10.21 MicrowaveEvent class diagram

3. **A method for the event to accept the listener** The abstract event has an abstract method `connectToListener` (Fig. 10.21), which the concrete events must implement. This method will be invoked by the context (or any 'switchboard') when it receives an event, and passes as a parameter the object that should be notified of the event.
4. **A method to send the listener to the event** This method resides in the 'switchboard'.
5. **The concrete listeners** Finally, we have the concrete classes that implement the specific listener interfaces, which in our example are the concrete states.

To demonstrate how the classes and interfaces will be used to eliminate conditionals, we examine the sequence diagram in Fig. 10.22, which shows what happens when we press the 'Cook' button while the current state is DoorClosedState. The sequence of actions is as follows:

1. The user clicks the 'Cook' button. This generates an instance of `ActionEvent` and control goes to the `actionPerformed` method of `GUIDisplay`.
2. The `actionPerformed` method notes that the event is a cook request, so it creates an instance of `CookRequestEvent`. It then sends the event as parameter to the `handleEvent` method of `MicrowaveContext`.
3. In the `handleEvent` method of `MicrowaveContext`, the parameter is used to call `connectToListener`. This is another polymorphic call. This ends up calling the `connectToListener` method of `CookRequestEvent`.

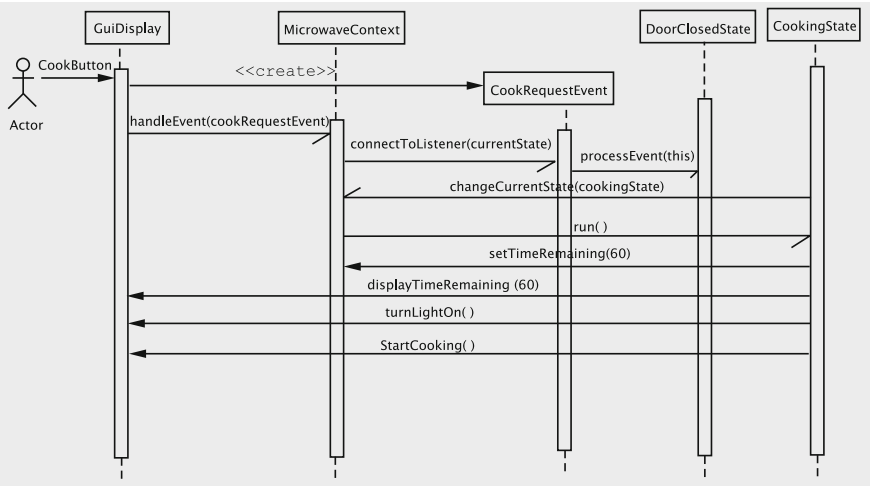

Fig. 10.22 Sequence diagram for cook request while in the DoorClosed state

4. The `connectToListener` receives the current state as parameter, which it uses to call the `processEvent` method of `DoorClosedState` (which is the current state) with `CookRequestEvent` as parameter.
5. The `processEvent` method changes the current state, which in turn modifies the display.

10.8.3 Implementation

The `MicrowaveEvent` class is abstract and extends `java.util.EventObject`. The `connectToListener` method is critical for making the switchboard function correctly.

```
import java.util.*;
public abstract class MicrowaveEvent extends EventObject {
  public MicrowaveEvent(Object object) {
    super(object);
  }
  public abstract void connectToListener(MicrowaveEventListener listener);
}
```

Each concrete event implements `connectToListener` and must invoke the appropriate method in the listener interface. However, *the parameter is explicitly known to implement only the general interface; we therefore cast the object to the specific listener interface and then invoke the method.* Since this cast may fail, the method is declared to throw the `ClasscastException`. The implementation of `connectToListener` for `CookRequestEvent` is shown below. Since not

all states need to handle the event, we might want to suppress message displays if a `ClassCastException` object is thrown.

Note that the `connectToListener` method needs to verify during execution that `listener` belongs to the class `CookRequestListener`. This verification is automatically done through downcasting and is therefore an application of RTTI.

```
public class CookRequestEvent extends MicrowaveEvent {
  public CookRequestEvent(MicrowaveDisplay display) {
    super(display);
  }
  public  void connectToListener(MicrowaveEventListener listener) {
    try {
      ((CookRequestListener) listener).processEvent(this);
    } catch (ClassCastException cce) {
    // message
    }
  }
}
```

The code for `MicrowaveEventListener` and one of its subinterfaces, `CookRequestListener`, are shown below.

```
public interface MicrowaveEventListener {
  public void processEvent(MicrowaveEvent event);
}
public interface CookRequestListener extends MicrowaveEventListener {
  public void processEvent(CookRequestEvent event);
}
```

The `handleEvent` method in the context casts `currentState` as a listener before it is passed; since the declared type of this variable is the abstract class `MicrowaveState`, such a cast would not be possible if we did not have the general listener interface.

The code has to reside in a `try` block for obvious reasons; to facilitate debugging, we add the `logException` method to the state classes.

```
public void handleEvent(MicrowaveEvent event) {
  try {
    event.connectToListener((MicrowaveEventListener) currentState);
  } catch (ClassCastException cce) {
    currentState.logException(cce);
  }
}
```

Partial code for the `CookingState` is shown below. Note that we no longer need to register or de-register and need to implement methods only for the events that this state is interested in.

```
import java.util.*;
public class CookingState extends MicrowaveState implements
       DoorOpenListener, CookRequestListener, ClockTickListener {
  // code for making the class a singleton
  public  void run(){
    context.setTimeRemaining(60);
    display.displayTimeRemaining(context.getTimeRemaining());
```

```
    display.turnLightOn();
    display.startCooking();
  }
  public void processEvent(DoorOpenEvent event) {
    context.changeCurrentState(DoorOpenState.instance());
  }
  public void processEvent(CookRequestEvent event) {
    context.setTimeRemaining(context.getTimeRemaining() + 60);
    display.displayTimeRemaining(context.getTimeRemaining());
  }
  // other methods
}
```

Before closing the subject, we should see some alternatives to downcasting when implementing the `connectToListener` method in the event classes. We could use the `instanceof` method to ensure that the listener indeed implements the appropriate interface. This approach is likely to execute faster since the overhead associated with the exception is avoided, but does not report situations where unexpected state–event combinations show up at runtime. Another alternative would be to have all states implement all listener interfaces, making methods to handle events that are not of interest to it no-ops; this would, however, mean that all states would have to be changed when new events are added, thus hurting reuse.

Eliminating Conditionals in `GUIDisplay`

As it stands, we still have conditionals in the `GUIDisplay` class. While a general approach to eliminating conditionals in the user interface might be tedious if not impossible, it turns out that we can eliminate them completely in `GUIDisplay`. The approach could be used to reduce the number of conditionals in user interface classes in general. The idea involves creating a separate class for each type of button. All such button classes extend some common functionality given in the class `GUIButton`.

```
import javax.swing.*;
public abstract class GUIButton extends JButton {
  public GUIButton(String string) {
    super(string);
  }
  public abstract void inform(MicrowaveContext context,
                              MicrowaveDisplay display);
}
```

The button for issuing the cook request is an instance of `CookButton` coded as below.

```
public class CookButton extends GUIButton {
  public CookButton(String string) {
    super(string);
  }
  public void inform(MicrowaveContext context, MicrowaveDisplay source) {
    context.handleEvent(new CookRequestEvent(source));
  }
}
```

Finally, the code in `GUIDisplay` simply calls the *inform* method on the source of the event.

```
public void actionPerformed(ActionEvent event) {
  ((GUIButton) event.getSource()).inform(MicrowaveContext.instance(),
   this);
}
```

10.9 Designing GUI Programs Using the State Pattern

Let us consider the execution of a typical GUI program. Initially, the program displays a window and waits for an input from the user such as a click on a button or selection of an item in a list box. The program processes this request and displays another screen, waiting, once again, for an input from the user.

When the program has displayed a window, we can view that as the current 'state' of the program. The event caused by user's actions such as button clicks and list selections, is the input to the state. Based on the state and the current input, the program takes some actions and makes a transition to another state, which displays yet another window.

Of course, a GUI program may not always wait for a user input when it displays a window. As an example, assume that the program presents a snapshot of a file system. The user may invoke a command to copy a set of files perhaps by dragging some icons or by doing a 'copy and paste'. The program may choose to display a simple window (a message box) with some message like 'copying in progress'. This window may not have any widgets for human use. The window disappears when copying is finished.

In the above case, the display of the message box represents the copying state of the program. The state's input is a notification from some copying code that copying has been completed.

10.9.1 Design of a GUI System for the Library

We briefly describe our approach to GUI design by building a GUI for the library system. The interface is more or less equivalent to the text-based interface we developed in Chap. 7.

Obviously, we need one screen for displaying the main menu, a screen for allowing the addition of books, yet another one for adding a member, and so on. Each of these screens corresponds to a state of the program.

Since there are too many states to be conveniently presented in a single diagram, we present our design in smaller parts.

The main menu and add books We have a state, MainMenu, for displaying the main menu. This has a number of buttons, each of which selects an operation on the library. When the 'Add Book' button is clicked, the system goes to the AddBook state. Since we anticipate that we will get a number of books to be added, we will

provide two buttons here: one to add a book and the other to signal that the screen can be dismissed. The state transition diagram is shown in Fig. 10.23.

Add member When we click the 'Add Member' button on the main menu, the system goes to AddMember, displaying a window to let the user input member details. After entering, the user may click the 'OK' button to enter the data into the library. The system displays the result in a new window (a new state), ShowResult, from which control goes to MainMenu when the user clicks the 'OK' button.

In AddMember, the user may also choose the 'Cancel' option to abandon the operation.

The details are shown in Fig. 10.24.

Issue books The state transition diagram is given in Fig. 10.25. As the reader may expect, the system goes to IssueBooks when the user clicks 'Issue Books' on the main menu. In this state, the user may enter the member id and click 'OK' or click 'Cancel' to return to the main menu. If a valid member id is entered, the system moves to the GetBookId state, which lets the user enter a book id. After entering each id, the user may click the 'OK' button to enter one more book id. Clicking 'Done' takes the system back to the MainMenu state.

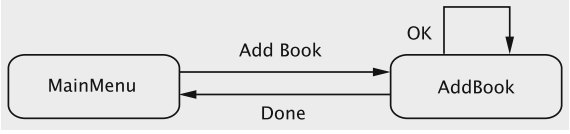

Fig. 10.23 Main menu and add books

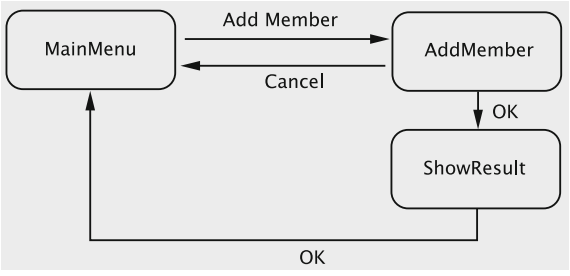

Fig. 10.24 Main menu and add member

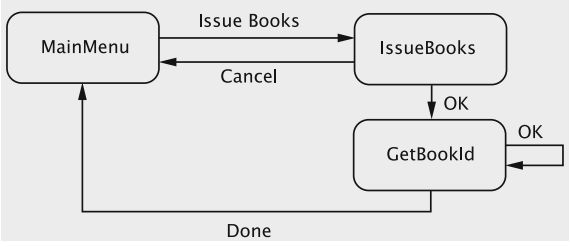

Fig. 10.25 Main menu and issue books

Return books and remove books The state transition diagrams for both cases are similar. After the appropriate choice ('Return Books' and 'Remove Books') is made from the main menu, the state is changed to either ReturnBooks or RemoveBooks. In each of these states, the user can enter a book id and click 'OK' or press 'Done' to return to the main menu. The state transition diagram is shown in Fig. 10.26.

Place hold and remove hold These two also have similar state transition diagrams and are shown in Figs. 10.27 and 10.28. We assume that explanations for the previous cases should provide enough clues to understand these cases.

Print transactions The flow of control is depicted in Fig. 10.29. Since a member may have had many transactions on a given date, the result is shown in a state called ShowLongResult.

Process holds The state transition is similar to 'Add Books' and is given in Fig. 10.30.

We leave the implementation of *Renew Books* as an exercise.

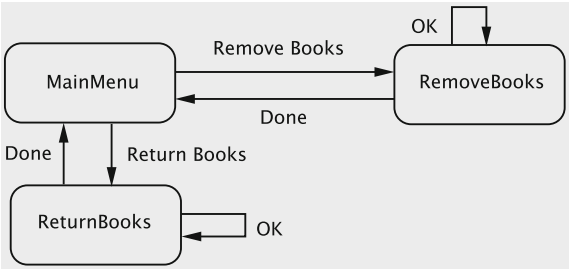

Fig. 10.26 Remove books and return books

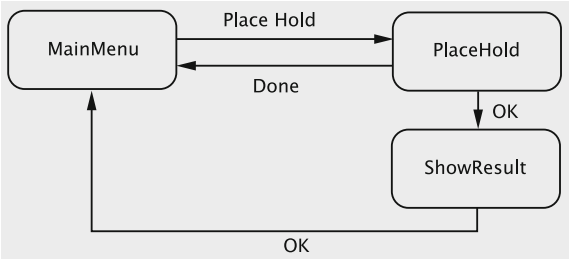

Fig. 10.27 Place hold

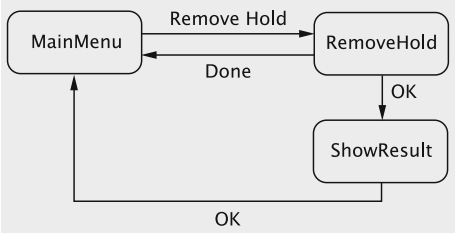

Fig. 10.28 Remove hold

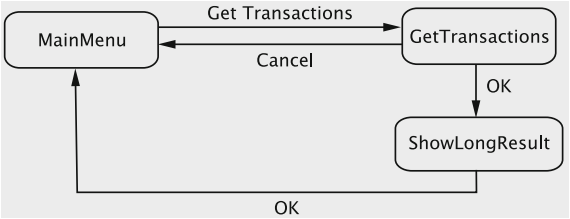

Fig. 10.29 Print transactions

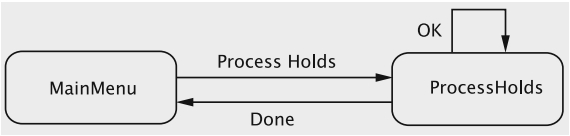

Fig. 10.30 Process holds

10.9.2 The Context

The system needs to transfer information between some pairs of states. For example, the member id entered in IssueBooks is to be used in GetBookId, and the transactions from GetTransactions must be given to ShowLongResult. This is achieved by means of a context, a singleton. The class diagram is shown in Fig. 10.31.

In the IssueBooks state, the user enters the member id clicks 'OK'. The state stores the member id in the context, retrieved by GetBookId. Similar transfers occur between some other pairs of states.

Implementation All states implement the interface `UIState`, which is given below.

```
public interface UIState {
  public void handle(Object event);
  public void run();
}
```

MainMenu is implemented as a `JFrame` and all other states are extensions of the Java class `JDialog`. Every dialog is modal, which ensures that main menu cannot be accessed until it is dismissed.

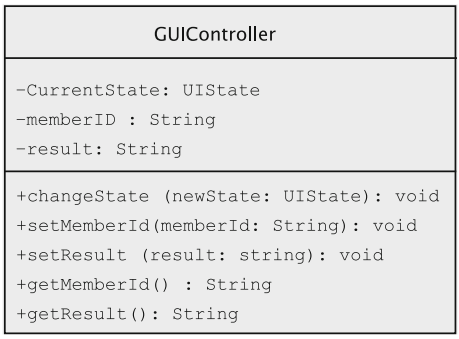

Fig. 10.31 The context

10.10 Discussion and Further Reading

10.10.1 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 [1]. 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 realised. 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 [2]. The context in this implementation provides methods to add states and transitions, which populate the transition table. These methods are accessed through a driver routine, thus allowing the context to be reused in other applications. Such an approach is particularly suitable for designing multi-panel interactive systems, such as the GUI for the library [3]. The approach that we have taken in the design of the microwave system lies somewhere in the middle of this range. The context stores the current state and also serves as a facade, but has nothing to do with the transition table.

10.10.2 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 behaviour 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 that they all conform to some common type.
3. One instance of each state class is created. In the case of the microwave oven, 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 `changeCurrentState` 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 centralised controller that uses the matrix (see Sect. 10.5.1) 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.

10.10.3 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 the 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 [4].

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 [5].

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 [6].

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 [7].

10.10.4 Recognising 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.

10.10.5 Handling the Events

We have discussed three mechanisms for dealing with events in this chapter. Creating an `enum` is the simplest, but does not allow us pass on any information along with the event and requires the use of conditionals. The standard event handling mechanism provided by Java requires that we have an event source that can register listeners that implement a custom interface and notify them when the event occurs. This requires the listener object to take full responsibility for ensuring that the connection is made. A third approach is to use a form of double dispatch where the source is aware of a switchboard that can connect the event to the listener. This approach can be generalised to other situations by having the event classes extend `EventObject` and the listener interfaces extend `EventListener`. The trade-off here is that we have an additional method call, but the listener does not have to do any extra housekeeping.

Events can be classified as **low-level** events, which represent window-system occurrences or low-level input, and **semantic** events which include all other events. A button click is a semantic event which is defined to occur when the mouse is pressed and released over the button's display. We could therefore achieve the same functionality by tracking the mouse movement and mouse clicks, which are low-level events. In general, it is preferable to listen for semantic events since there may be alternative mechanisms (such as key sequences) that can activate a button.

In addition to the features presented in the chapter, Java provides some other features for custom events. Notable among these is the facility for directly manipulating the system event queue. This can be useful in situations where events have to be added, removed, or bypassed, and also in situations where update of graphical components is involved. Although it is not particularly useful for the kind of systems we have presented here, we shall go through the basic steps of this process using our microwave as an example.

To be placed in the queue, the event must extent `AWTEvent` instead of `EventObject` as we did earlier. To define a subclass of `AWTEvent`, we need to give an unused *event ID number* to the superclass. These details are shown for the `CookRequestEvent` below. (The choice of `1111` is arbitrary.)

```
class CookRequestEvent extends AWTEvent {
  public CookRequestEvent(CookRequestManager manager) {
    super(manager, COOK_REQUEST_EVENT);
  }
  public static final int COOK_REQUEST_EVENT =
                          AWTEvent.RESERVED_ID_MAX + 1111;
}
```

Next, we need a way to actually post the event on the event queue. This is done in the `actionPerformed` method. Note that the source is cited as the `CookRequestManager` instance. This is because the system automatically calls the `processEvent` method of the source to dispatch the events, unlike the earlier case where `processEvent` was explicitly called from `actionPerformed`.

```
public void actionPerformed(ActionEvent event) {
  EventQueue queue = Toolkit.getDefaultToolkit().getSystemEventQueue();
  if (event.getSource().equals(cookButton)) {
    queue.postEvent(new CookRequestEvent
                    (CookRequestManager.instance()));
  }
  // code to process other events
}
```

Finally, we have the code for dispatching the events. The method `processEvent` must ensure that it has the right kind of event before it notifies the listeners.

```
public class CookRequestManager extends JComponent {
  // other fields and attributes
  public void processEvent(AWTEvent event) {
    if (event instanceof CookRequestEvent) {
      CookRequestEvent cookRequestEvent = (CookRequestEvent) event;
      EventListener[] listeners
            = listenerList.getListeners(CookRequestListener.class);
      for (int index = 0; index < listeners.length; index++) {
        ((CookRequestListener)listeners[index]).cookingRequested
                                              (cookRequestEvent);
      }
    } else {
      super.processEvent(event);
    }
  }
}
```

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 behaviour 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 Chap. 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 minutes; the cycle repeats.
(c) Air-conditioner: The air-conditioner immediately turns on. If the room temperature is too high, it operates the air-conditioner until the room temperature hits the set temperature.
(d) Heater: The heater immediately turns on. If the room is too cold, it operates the 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.

10.11 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. Modify the UI for the microwave so that the buttons OpenDoor and CloseDoor are replaced by a single `JSlider`. Discuss the simplicity/difficulty of effecting such a change for all the event processing options discussed in the chapter.
4. 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.
5. Draw sequence diagrams to document the flow of control for the microwave system for each of the following cases:

   (a) When the Java event framework is used.
   (b) When the context is used as a switchboard.

6. 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 s 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 s.

(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.

7. In the GUI implementation of the library system, draw the state transition diagram for renewing books and write the corresponding code.
8. Rewrite the code for the `connectToListener` method using the alternative approaches discussed in the text.

References

1. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, Boston, 1994)
2. T. Cargill, *C++ Programming Style* (Addison-Wesley Professional, Reading, 1992)
3. B. Meyer, *Object-Oriented Software Construction* (Prentice Hall, Upper Saddle River, 1997)
4. Unknown, The observer pattern. Core Java technologies. Technical tips. http://java.sun.com/developer/JDCTechTips/, January 2006
5. B. Goetz, Java theory and practice: be a good (event) listener. guidelines for writing and supporting event listeners. http://www.ibm.com/developerworks/, July 2005
6. D. Gruntz, Java design: on the observer pattern. Technical report, University of Applied Sciences, Aargau (2004)
7. A. Holub, Programming java threads in the real world, part 6: the observer pattern and mysteries of the awteventmulticaster. http://www.javaworld.com/javaworld/jw-03-1999/jw-03-toolbox.html, March 1999