JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 2, March-April 2003

Pattern Integration: Emphasizing the De-Coupling of Software Subsystems in Conjunction with the Use of Design Patterns

Sarnath Ramnath, Department of Computer Science, St. Cloud State University, Minneapolis, U.S.A.
Brahma Dathan, Department of Information and Computer Sciences, Metropolitan State University, Minneapolis, U.S.A.

1 INTRODUCTION

Decoupling and reuse issues must be emphasized in classroom examples in Junior and senior level software classes on software construction, especially on object-oriented design because of the wide use of that paradigm in modern computer systems. Microsoft's Component Object Model (COM) and Java Beans are good examples of products that build on these principles.

Obviously, any object-oriented design class must discuss examples using the notion of Design Patterns. Two patterns that we have had occasion to discuss in the classroom are the Command pattern and the State pattern. In one possible organization of an object-oriented design course, one would briefly discuss the concepts of reuse and decoupling before introducing design patterns and related examples. The examples should impress upon the student the benefits of reuse and decoupling.

In this short paper, we discuss some of the issues we have come across regarding decoupling and reuse while teaching the State and Command patterns. We discuss some of the ideas we have used in the classroom. In our experience, our procedure has made the presentations more effective.

2 TWO EXAMPLES

Example 1: State Pattern: The one-minute microwave

This is an example from Practical Object-Oriented Development with UML and Java by Lee and Tepfenhart. The one-minute microwave is a simple system with the following requirements:

1. There is a single button available for the user.
2. If the door is closed and the button is pushed, the oven will be energized for one minute.
3. If the button is pushed while the oven is energized, the cooking time is increased by one minute.
4. If the door is open, pushing the button has no effect.
5. The oven has a light that is turned on when the door is open, and also when the oven is cooking. Otherwise the light is off.
6. Opening the door stops the cooking and clears the timer (i.e., remaining cooking time is set to zero).
7. When the cooking is complete (oven times out) a beeper is sounds and the light is turned off.

As explained by the authors of the text, this system can be modeled using Finite State Machines (FSM). Note the Oven and the Timer perform independent functions: the Timer keeps track of the time remaining, and counts down with every clock-tick. The oven turns the powertube on/off, turns the light on/off etc. Separating the Timer from the Oven also enables us to modify the design of each one independently. All that the Oven needs to know is when to turn itself on and off. (Separating the Timer allows us to introduce additional functionality; for instance we could incorporate a wait feature that allows the user to turn the oven on after a specified waiting period, which would be dealt with by the timer informing the oven to turn itself on at a designated time. Since this functionality is provided by the timer alone, separating the two makes design sense.)

As a result, we have two Finite State Machines (FSMs) operating in tandem. The Timer FSM could be in one of three states : (1) The Sleep state where it ignores both clock ticks and and the button, (2) the Idle state where it ignores clock-ticks, but listens to the button and (3) the Active state where it listens to both button and clock-ticks. When time remaining drops to zero, the oven must be notified.

The Oven FSM also has three states: (1) The OpenDoor state when it is waiting for the door to be shut, (2) the Idle state where the door is closed but the Oven is off and (3) the Active state where the Oven is operating.

This set-up suggests the use of the State Pattern with two Controllers, each holding the transition table for the corresponding FSM. As is the usual practice in the implementation of the State Pattern, the Controller observes the state that is currently active (the Current state of its FSM) and is notified, along with the appropriate transition code, whenever an event occurs. The controller then switches to a new Current state in accordance with the transition table.

Clearly, to improve reuse, Timer states should not be directly aware/dependent on the Oven states and vice-versa. However, there is a need for time states and oven states to communicate. The Active Timer state must notify the Active Oven state when time remaining drops to zero. (Likewise, if we add a wait feature, the appropriate Timer state must notify the Idle state of the Oven.) This results in a situation where there are two subsystems and objects inside one subsystem which need to communicate with an object inside the other.

The problem is how to design the system that minimizes coupling and increases the chance of reuse.

Now let us look at a second problem: the one that uses the Command pattern.

The Model View Controller (MVC) Paradigm.

In this paradigm, the View subsystem creates a number of widgets that make up a GUI interface with which the user interacts with the system. The Model subsystem maintains the data related to the GUI interface; for example, data associated with a table. The controller orchestrates the interactions between the view and the model subsystems.

A typical design would have the widgets storing references to model objects. User interaction (for example, clicking of a button), will result in the view (specifically, the widget) invoking some action that changes the model.

The Command pattern is especially useful in the Model View Controller paradigm. The actions that change the model can be implemented through Command objects. Once again, we have a situation where the objects inside the View subsystem (viz., widgets) need to communicate with specific objects outside this subsystem (viz. commands). A common implementation that several textbooks use to achieve this functionality is to maintain a reference to a Command object within the corresponding widget.

However, maintaining such references within the widgets themselves reduces the use of reuse. One invariably ends up with some piece of the system that is aware of the internal details of the view subsytem and the individual commands.

3 THE IDEA OF A SWITCHBOARD

The two problems we discussed in the previous section had a common theme. In the abstract we have two subsystems with objects within one subsystem having to communicate with objects in the other subsystem. We propose the use of a separate component, which we call a switchboard, to maintain the mapping between these objects.

Let us first look at a solution to the implementation of the MVC paradigm because the solution is simpler. To aid the discussion, we use an example in the Java programming language.

Assume that we have a view that contains two widgets, both buttons. Let us say that one button is for opening and the other is for closing a file. In our implementation, there are two classes, `FileOpenButton` and `FileCloseButton`, both extending the `JButton` class. Both buttons maintain a reference to the switchboard object; and, in particular, they do not reference to Command objects: objects that provide the necessary functionality for opening or closing.

When the user clicks on one of the buttons, say, the instance of the `FileOpenButton`, the button generates an instance of the Java event class, `ActionEvent`. In our design, this event is caught by the button itself. The button then informs the switchboard that the button has been clicked, which is accomplished by calling a method, `processFileOpen`, of the switchboard.

Here are the above ideas in Java:

```
public   class   FileOpenButton   extends   JButton   implements
ActionListener {
   SwitchBoard switchBoard;
   public FileOpenButton(SwitchBoard switchBoard) {
      this.switchboard = switchboard;
      addActionListener(this);
      // other code
   }
   public void actionPerformed(ActionEvent event) {
      switchBoard.processFileOpenEvent();
   }

   // other methods; for example, methods for appearance changes
}
public   class   FileCloseButton   extends   JButton   implements
ActionListener {
   SwitchBoard switchBoard;
   public FileCloseButton (SwitchBoard switchBoard) {
      this.switchboard = switchboard;
      addActionListener(this);
      // other code
   }
   public void actionPerformed(ActionEvent event) {
      switchBoard.processFileCloseEvent();
   }

   // other methods; for example, methods for appearance changes
}
```

We now elaborate the idea behind the methods `processFileOpenEvent` and `processFileCloseEvent`. As noted earlier, `JButton` objects generate `ActionEvent` objects. The method processFileOpenEvent is specifically created to handle the ActionEvent objects generated from the instance of FileOpenButton; similarly, the method processFileCloseEvent caters to the events generated by the instance of the FileCloseButton.

In a more abstract sense, assume that we have an instance of a widget class WX, where X is some meaningful name such as Button or MenuItem in the view. As an example, W could be FileOpen and X could be Button. Suppose that this widget is capable of generating an event of type E (example: ActionEvent), and that we desire to handle these events. It is convenient for pedagogical purposes to think of objects of type E arising from instances of WX as belonging to events of a class WEvent. The switchboard has a method processWEvent() that corresponds to WEvent. The widget calls processWEvent() when it recognizes an event of type WEvent.

The above discussion showed how events from the view are communicated to the switchboard. Obviously, these events must be transferred to the command objects. This issue is described below. Let us continue with the example of opening and closing files. Assume that we have a class FileOpenCommand that provides the capability to open files.

An instance of the FileOpenCommand class, when created, registers itself with the switchboard. Essentially, it informs the switchboard that it is available to handle open file commands from any view. To communicate this event, the switchboard provides a method, registerFileOpenCommand(Command).

The following code shows the central ideas behind the Command class.

```
public class FileOpenCommand implements Command {
   SwitchBoard switchboard;
   public FileOpenCommand(SwitchBoard switchboard) {
      this.switchboard = switchboard;
      switchboard.registerFileOpenCommand(this);
   }
   public void execute() {
      openFile();
   }
   public void openFile() {
   // code to open a file
   }
   // code for other functionality such as undo, redo, etc.
   }
```

The code for FileCloseCommand would be similar.

Thus, corresponding to the event WEvent described earlier, the switchboard has a second method registerWCommand.

We give below the code for the switchboard.

```
public class SwitchBoard {
   Command fileOpenCommand;
   Command fileCloseCommand;
// references to other Command objects
   public void registerFileOpenCommand(Command command) {
      fileOpenCommand = command;
   }
   public void processFileOpenEvent() {
      if (fileOpenCommand!= null) {
         fileOpenListener.execute();
      }
   }
   public void registerFileCloseCommand (Command command) {
      fileCloseCommand = command;
   }
   public void processFileCloseEvent() {
      if (fileCloseCommand!= null) {
         fileCloseCommand.execute ();
      }
   }
// Methods to register other Command objects and transfer events
}
```

The view creates the widgets and the switchboard. The controller works as follows:

```
Switchboard switchboard = view.getSwitchboard();
Command fileOpenListener = new FileOpenCommand(switchboard);
Command fileCloseListener = new FileCloseListener (switchboard);
```

The sequence of events/methods invoked would be as follows:

User clicks Fileopen button → `FileOpenButton.ActionPerformed()` →

`switchboard.processFileOpenEvent()` → `FileOpenCommand.execute()`

Note that this setup allows us to design and modify the subsystems independently of one another. As formulated above, the listeners that are registered by the switchboard are command objects. In general, this need not be the case. We shall now discuss the solution for the microwave problem using the above abstraction that does not place any restriction on the nature of the listeners.

Solution to the One-Minute Microwave

Following our abstraction, the Timer subsystem would maintain a switchboard that generates events corresponding to all the messages that the Timer has to send out, i.e., Time remaining equals zero, wait time has elapsed, etc. The switchboard reference is available to the Oven, and the appropriate Oven states can then register themselves to listen to these events. (For instance, the Active state would listen for the time remaining equals zero event and the Idle state would listen for the wait time elapsed event.)

To produce a simulation of such a system, we need some way of listening to the physical events, viz., opening door, ticking of the clock etc., each of which is defined as an interface.

```
public interface ClockTickListener{
   public void clockticks();
}
```

As dictated by the State Pattern, every state of the timer FSM must extend an abstract `TimerState`.

```
public class TimerWaitState extends TimerState implements
                                             ClockTickListener {
   SwitchBoard switchBoard; int waitTime;
   public TimerWaitState(SwitchBoard switchBoard) {
      this.switchboard = switchboard;
      // other code
   }
   public void clockticks() {
      if (waitTime != 0) {
         waitTime--;
         if waitTime == 0
               switchBoard.processWaitElapsedEvent();
      }
   }
// other methods; for example, methods for setting/restting the
// timer
}
```

Likewise we have other `TimerStates` - where the Timer is active, idle, etc. - each of which may generate events through the switchboard. As before the switcboard needs methods to register listeners and process events. Unlike the previous example where the listeners were `Commands`, the type of the listeners may not be known to the author of this switchboard. Accordingly, we define an interface for each event that is generated within the subsystem. The `WaitElapsedListener` is the interface that must be implemented by any object that wishes to be notified when the wait time elapses.

```
public interface WaitElapsedListener{
   public void waitTimeOver();
}
```

The switchboard is as follows:

```
public class SwitchBoard {
        WaitElapsedListener  waiter;
  // references to  listeners for the other Timer events.
     public void registerWaitListener(WaitElapsedListener listener) {
        waiter = listener;
     }
     public void processWaitElapsedEvent() {
        if (waiter != null) {
           waiter.waitTimeOver();
        }
     }
  // Methods to register other listeners and transfer other events
  }
```

The Oven states extend the abstract `OvenState` class, and implement the interfaces needed for the events. The `IdleOvenState`, for instance, wishes to be notified when the wait time elapses.

```
  public class IdleOvenState extends OvenState implements
              WaitElapsedListener {
     SwitchBoard switchboard;
     public IdleOvenState(SwitchBoard switchboard) {
        this.switchboard = switchboard;
        switchboard.registerWaitListener(this);
     }
     public void waitTimeOver() {
     // code to start the Oven
     }
     // code for other functionality such as door opening,
     // button being pushed etc.
  }
```

In addition each state may implement other interfaces. For instance the `IdleOvenState` also has to listen to the button being pressed, door being opened, etc. Using the event paradigm here allows the responses to all these events to be defined independently.

The `OvenController` creates all the oven states and sets up the transition table for the FSM.

```
  public class OvenController{
     public OvenController (Switchboard switchboard){
        OvenState S1 = new IdleOvenState(switchboard);
     // other code to create states and set up transitions
     }
  }
```

Once again, we can see how the switchboard and the events allow us to design and modify the two subsystems (FSMs) independently, within the context of a Design Pattern. What this gives us is a general solution to the problem of tight-coupling between subsystems that can be presented in a senior-level undergraduate course. The examples are simple, and can be found in current text books. By employing the switchboard in more than one context, the students understand the importance of avoiding tight coupling. Since the same solution can be employed in more than one context, the presentation is somewhat easier and can be done with less overhead.

4 DISCUSSION, FUTURE WORK, AND CONCLUSIONS

As we have pointed out, we have found that many textbooks introduce examples on design patterns without adequately stressing decoupling. Usually, examples show the view object maintaining a reference to the command object. This close dependency makes it a little awkward for the instructor to fully justify the design.

Our approach requires further exploration. The following observations suggest possible avenues for future work.

1) Registering multiple listeners. We feel that examples that can demonstrate this would be somewhat complicated and therefore unsuitable for instructional use. However, the above design can be easily modified to accommodate this by maintaining a separate list for each kind of listener. The switchboard would then invoke the appropriate method for each listener, just like the event handling methods in the JComponent class.
2) Reducing the overhead of the additional function call. The switchboard results in one additional function call for each event. It is desirable to eliminate this overhead.
3) The switchboard has a behavior similar to a Mediator, particularly in the MVC example. In a more general setting, as seen with the State Pattern, it appears to be more closely coupled with the subsystem that generates the events, and is thus more like a Facade.
4) The switchboard is ideally implemented using the Singleton Pattern. This idea may be introduced at a later point in the course.
5) The system could have several sub-systems that generate events; in that case, each sub-system has its own switchboard class. The subsystem(s) that listen(s) to the events can get the appropriate switchboard(s) using the Singleton Pattern.

About the authors

Sarnath Ramnath recieved his B.Tech and M.Tech degrees from the Indian Institute of Technology, New Delhi in 1984 and 1987 respectively, and his PhD in Computer Science from SUNY, Buffalo in 1994. His areas of interest include Algorithm Analysis and Design, Data-structures, Computational Geometry and Object-oriented Software Design. He is currently Professor and Chair of the Dept of Computer Science at Minnesota State University, St Cloud.

Brahma Dathan holds a Ph.D. degree in Computer Science from the University of Pittsburgh. He is currently on the faculty at Metropolitan State University in Minneapolis, Minnesota. Previously, he was on the faculty at the University of Wyoming and Saint Cloud State University. His interests are in the areas of object-oriented systems, distributed systems, and database systems.