CHAPTER 3

The Game of Life

This chapter provides an in-depth look at an implementation of John Conway's Game of Life—probably the most widely implemented application on the planet. You'll look at this particular program because my version applies ten distinct design patterns, all jumbled together as they are in the real world. At the same time, the program isn't so large as to be impossible to understand.

I can't really devote enough space in this chapter to give Life it's due, but reams have been written on the subject. I've set up a web page at `http://www.holub.com/software/life/` that lists various links to Life resources and also provides an applet version of the game (written by Alan Hensel). You can find the source code for the implementation discussed in this chapter on the same web page.

I strongly recommend you play with the game before you continue; otherwise, a lot of what I'm about to talk about will be incomprehensible.

My implementation of Life uses the Java client-side GUI library (Swing) heavily, and I'm assuming some familiarity with that library. You don't need to be an expert Swing programmer, but I'm assuming you know the basics. If you've never used Swing, you should work through the Swing Tutorial on the Sun web site (`http://java.sun.com/docs/books/tutorial/uiswing/`).

This chapter has a lot of code in it. I don't expect you to read every line—I've called out the important stuff in the text. I'm often frustrated by books that don't show entire programs, however. It seems like the code I'm interested in is never there. Consequently, I've risked putting too much code in the text in order to show you the complete program. Feel free to skim if you're bored or overwhelmed by the sheer volume of the stuff.

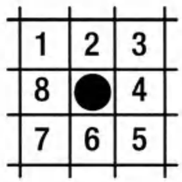
Finally, the code in this chapter is toy code (not the case with the SQL interpreter in the next chapter, which is production code). Consequently, I let myself get rather carried away with the patterns. The point of the exercise it to learn how design patterns work, however, not to write the best possible implementation of Life.

A. Holub, *Holub on Patterns: Learning Design Patterns by Looking at Code*

Get a Life

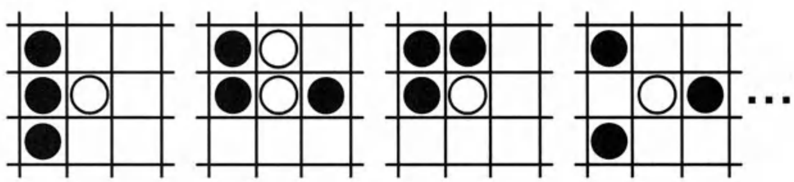
Life is a simple cellular automaton like the ones discussed in Chapter 1. Among other things, Life models organic patterns of behavior—cell growth in a petri dish or embers in a fire, for example. It can also behave in interesting programmatic ways. You can, for example, make a Life game behave like a Turing machine (so in theory, it could mimic any computer.) An anthropologist friend of mine says that some of the patterns remind her of behavioral patterns within human societies. Life also demonstrates "emergent" behavior—the behavior of the system as a whole can't be predicted solely by looking at the behavior of the objects that comprise the system (and is much more interesting than the component-level behavior).

The standard game board is a large rectilinear grid. Each cell has eight neighbors.

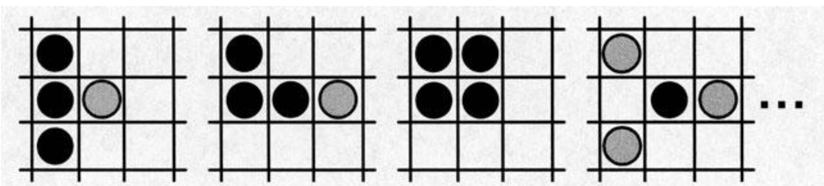


A cell is either "dead" or "alive." You "seed" the game by marking cells as alive, and then you set things going. Two passes are made every time an internal clock ticks. In the first pass, the cells determine their next state by examining their neighbor's state. In the second pass, the cells transition to the previously computed state. Here are the rules:

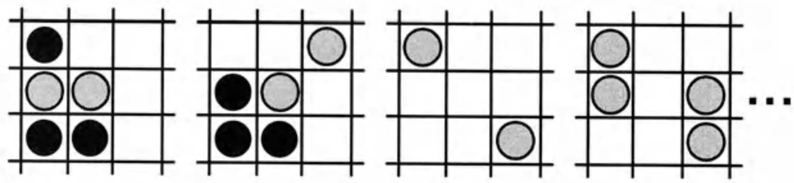
- A dead cell comes alive when it has exactly three live neighbors. In the following examples, the cells containing black dots are alive, and the cells marked with hollow circles will come alive on the next tick.



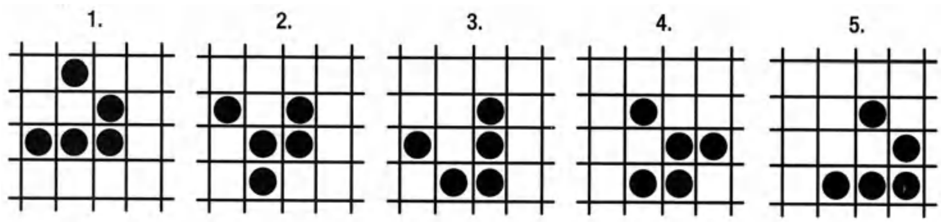
- A cell stays alive if it has exactly two or three neighbors. In the following examples, the cells containing black dots will stay alive.



- Otherwise, the cell dies from either loneliness or overcrowding. In the following examples, the gray cells will die on the next tick (the example on the left from overcrowding, the others from loneliness).



That's it. Not much in the way of rules, but depending on how it's seeded, the game board exhibits remarkable, very lifelike, behavior. The simplest example of interesting behavior is a *glider*, demonstrated by the following seed state (frame 1, on the left) and four subsequent game states. After the first two ticks (in frame 3), the glider has flipped itself symmetrically (along the diagonal axis) and moved itself down one row. After the fourth tick (frame 5), it flips back and moves over one column. It's now back in its original configuration but is offset diagonally by one cell from the starting position. Since the original configuration of cells is now restored, the pattern repeats indefinitely, and the group of cells glides toward the lower-right corner of the screen. When you look at the screen, you tend to think of the glider as an object that's moving across the board, but that's not what's going on at all. The real situation is just cells turning themselves on and off based on their neighbor's state. The cells have no notions at all of gliders or of what's happening on the board as a whole.



Hensel's Life applet at `http://www.holub.com/software/life/` demonstrates a bunch of interesting behaviors. Click the Empty Universe button and then click Open to open a catalog of preseeded Life games. The other buttons on the page bring up and run preseeded Life games that demonstrates a few of the more interesting patterns.

Charting the Structure of Life

I've sketched the static structure of my implementation of Life in Figure 3-1. Though I'd normally start designing with the dynamic model, I've found that when you're trying to understand (rather than design) an application, a good grasp of the static structure of the system is an important precursor to drilling into the messaging. I'll talk about the dynamic model as I drill into the patterns.

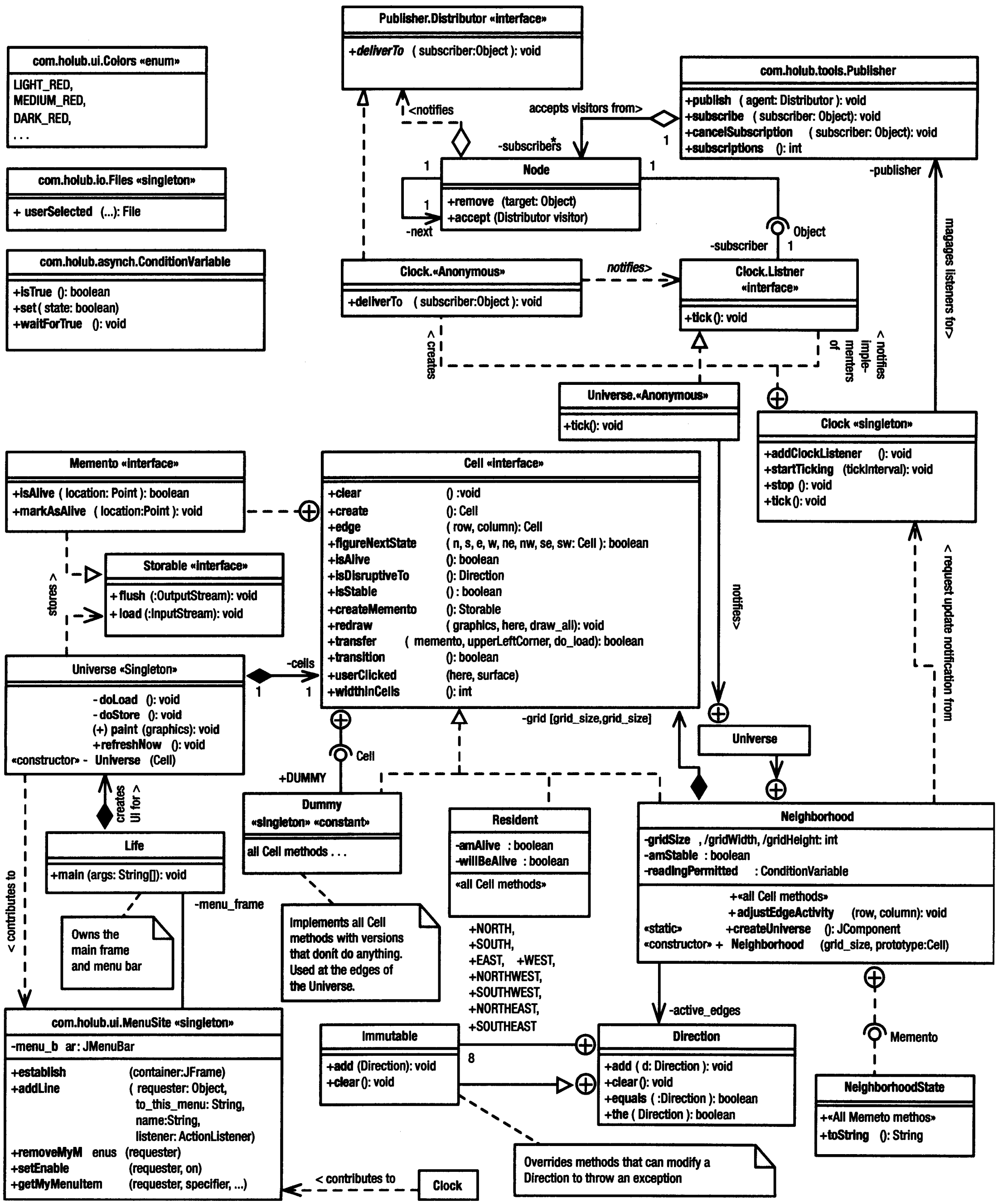


Figure 3-1. *The static structure of Life*

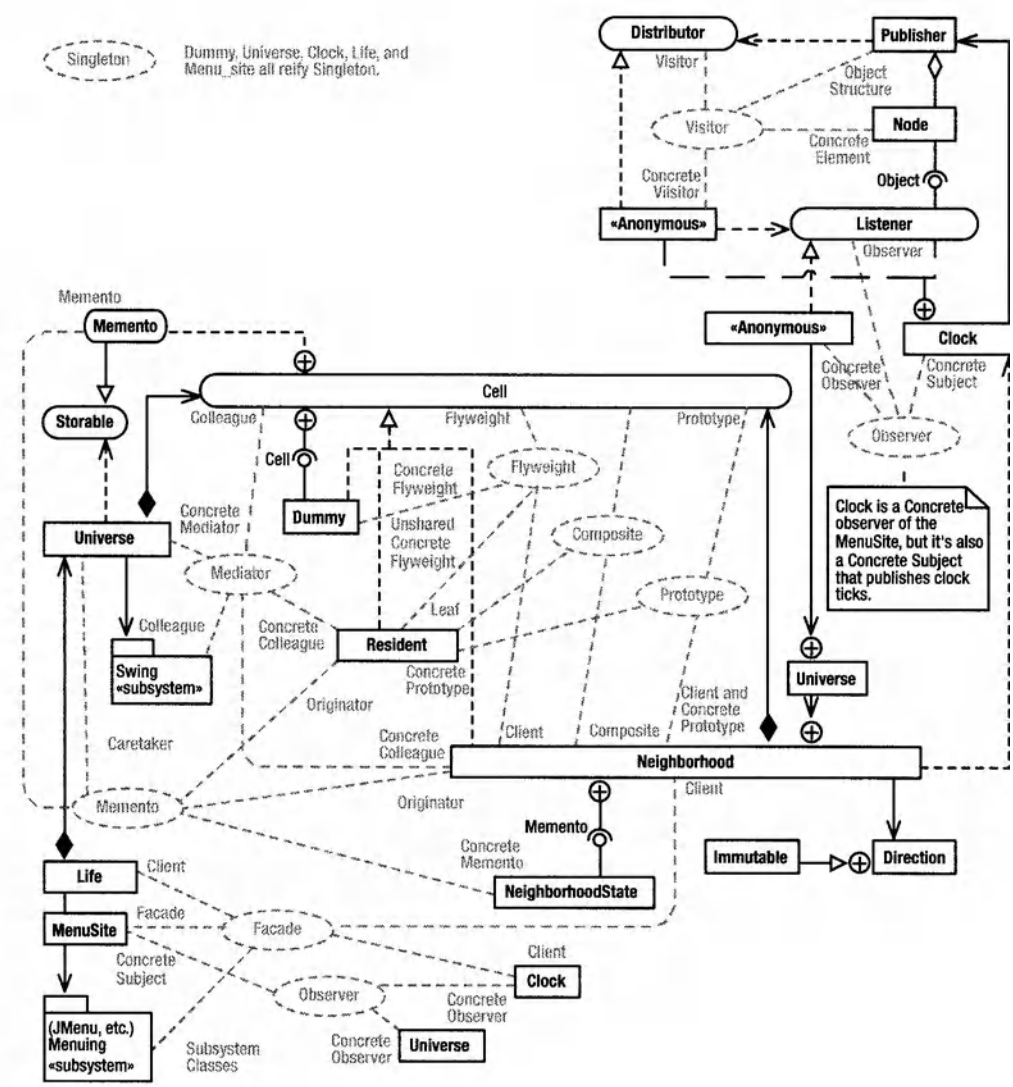


Figure 3-2. *The design patterns in Life*

Figure 3-1 shows a lot of classes, but I'll present them in small doses so you can see how they work together. Perhaps more interesting than the class diagram is Figure 3-2, which shows the static structure with the extraneous details stripped out and the main design patterns identified. I've put the interfaces into lozenges so that you can pick them out easily. I've also tried to keep the classes more or less in the same relative positions as they are in Figure 3-1, so you can correlate the two diagrams easily. You may want to bookmark these two diagrams; I'll be referring to them regularly. I call this variant of a class diagram a *patterns diagram*, which is not an official UML term. I find patterns diagrams to be quite useful in understanding a program's structure.

This diagram has a lot of design patterns—nine significant ones—all jumbled up in complicated ways. The `Cell` interface, for example, participates in four design patterns. The `Neighborhood` class participates in seven patterns! This is not the neat picture you'd expect from the Gang-of-Four book, but it represents the real world pretty accurately. Though Figure 3-2 looks like so much spaghetti, we'll take it one pattern at a time. (The real situation is even worse—I've omitted several "building-block" patterns from the diagram because it was already too cluttered.)

Don't panic.

When I first showed these diagrams to my wife Deirdre, who's also a programmer, her initial reaction was "that's so complicated I don't want to deal with it." Once we started going through the system, as I'll do with you as the chapter unfolds, her reaction changed to "this diagram is really rich." By "rich," she meant that the drawing conveys a lot of useful information in a compact form—it's dense. Density in design documents is good. A knowledgeable reader can glean an enormous amount from Figure 3-2 in a glance; this same information would take hours to convey without the vocabulary supplied by the patterns.

That transition, from "complicated" to "rich" is an important one and is typical of what happens when you start being able to apply the patterns with ease. The patterns let you make sense of the overall structure, so the appearance of complexity falls away along with the concomitant confusion. The incomprehensible becomes clear. As a client of mine once said, "I don't see how people can possibly program OO without a picture in front of them."

Figure 3-2 contains other interesting facets. For example, Flyweight, Composite, and Prototype (all in the middle of the figure) are almost identical structurally. The same three classes, along with their associated relationships, participate in all three patterns. If all you had was the static structure, you'd be confused, since the structure could indicate any of the three patterns, or perhaps none of them. Simply because you have a certain structure doesn't mean that you have a pattern. My point is one I made in Chapter 1—you can't identify a design pattern solely by static structure. You have to know the intent of the designer as well. Also, note how the patterns overlap. The notion of pattern cut-and-paste is nonsensical on its face—patterns just don't occur in the sort of splendid isolation that allows a clean paste operation.

The Clock Subsystem: Observer

Now let's look at the code itself. I'll start describing the classes at the edges of the system—looking at the ancillary pieces used by the core abstractions. These pieces form stand-alone subsystems, so they're easy to look at in isolation.

You'll see the clock subsystem in the upper-right corners of Figures 3-1 and 3-2. The first pattern it uses is **Observer**. `Clock` uses Observer to fire periodic clock-tick events at interested parties (in this case, the `Universe` via an anonymous inner class).

Observer is also used in Java's menuing system, which I'll need to talk about anyway, so I may as well cover it now. Figure 3-3 shows Java's menuing system.

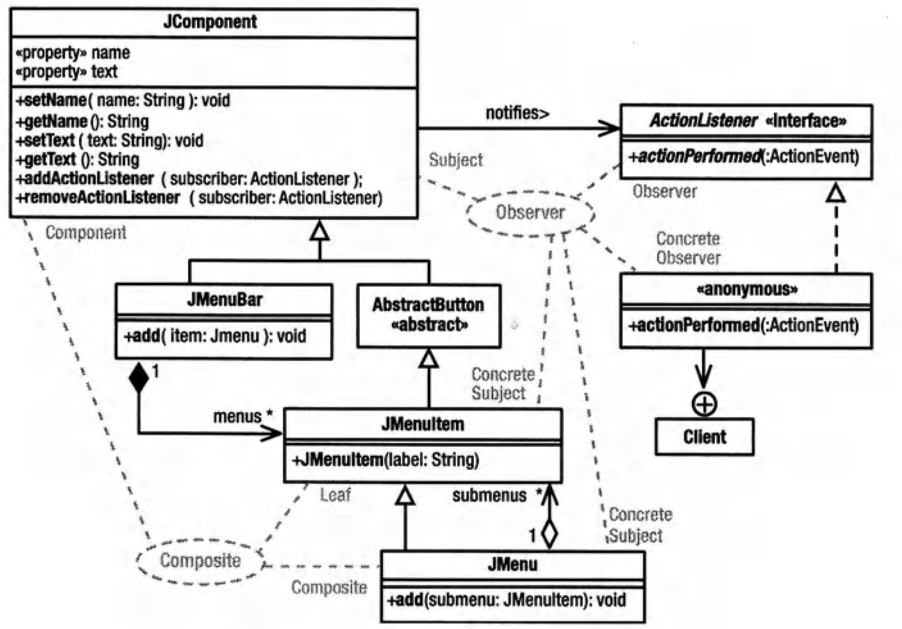


Figure 3-3. *Java's menuing subsystem (simplified)*

The main intent of Observer is to decouple an object that's interested in some event from the originator of that event. In the menuing system, the event occurs when you click a menu item, and whoever is interested in that event needs to find out when the item is selected.

The best way to see the notification-related problems that Observer solves is to look at the wrong way to do it: an implementation-inheritance based solution.

```
class BadJMenuItem
{   abstract void itemSelected();
}

class myMenuItem extends BadJMenuItem
{   public void itemSelected()
    {   // do whatever you'd do when the item is selected.
    }
}
```

This approach has two difficulties:

- You have to derive a class from `BadJMenuItem` for every menu item in the system, perhaps requiring hundreds of classes, all of which could be fragile.
- When a menu item is selected, you can notify only objects in the visual subsystem (in other words, `BadJMenuItem` derivatives). More often than not, the object that needs to be notified is some "business object" in the program, however. (In the case of Life, the `Clock` object needs to be notified when the user selects a new clock speed.) Passing the

notification through a visual object to get it to the party that's actually interested is needless work, and the unnecessary complexity of the intermediary class creates a maintenance problem. I'll discuss this issue further in a moment.

Observer—the pattern the real `JMenuItem` uses for notification—decouples the object interested in the event (the Observer) from the object that sends the notification (the Notifier, called the *Subject* by the Gang of Four for reasons that are completely mysterious to me). This pattern is also called *Publish/Subscribe*, which is a convenient metaphor for what happens. A publisher sends publications to a list of subscribers. Subscribers must subscribe to the publications by sending a message to the publisher, and subscribers can cancel their subscription at any time.

In the reification of Observer in Figure 3-3, the `JComponent` (or one of its derivatives) is the Subject/Publisher. That is, `JMenuItem` or `JMenu` can both take on the Subject (or Publisher) role. The `ActionListener` interface has the role of Observer (or Subscriber), and the implementing class has the role of Concrete Observer/Subscriber.

A Subject publishes notifications by sending them to Concrete Observers as an argument to some method of the Observer interface. Concretely, a `JComponent` publishes `actionEvents` by sending them to `ActionListeners` as an argument to the `actionPerformed(...)` method.

Here's the code that sets up a simple Observer that's notified of menu-item selections:

```
class Subscriber implements ActionListener
{   public void actionPerformed(ActionEvent e)
    {   // do whatever you do when the menu item is selected.
    }
}
```

You "subscribe" like this:

```
JMenuItem lineItem = new JMenuItem("Foo");
//...
lineItem.addActionListener( new Subscriber() );

JMenuBar menuBar = new JMenuBar();  // add the item to the menu bar
menuBar.add( lineItem );
```

Thereafter, when the user selects the menu item, the Subject notifies the Concrete Observer by calling one or more of the methods in the Observer interface. In concrete terms, the `JComponent` (the publisher) sends notifications to the its `ActionListeners` (the subscribers) by sending each of them an `actionPerformed()` message. It's important to note that the menu bar on which the menu item resides is not involved in the notification process. Notifications go directly from the publisher to the subscriber. This way you don't need to create and maintain intermediary "mediator" objects that do nothing but relay messages.

The current `JMenuItem` subscriber is little more than a Command object (discussed in Chapter 2) that's passed into the Publisher, which invokes the methods of the Observer interface to send an event to a subscriber. The code that actually notifies the subscribers is encapsulated in the Command object.

The `JComponent` class implements the publication mechanism by keeping a list of subscribers. (Other reifications may implement the subscription mechanism in the Concrete

Subject, in which case `JComponent` could look more like the classic Gang-of-Four reification where the Subject is an interface.)

A more realistic example of Observer uses an anonymous inner class as the Concrete Subject/Subscriber.

```
class Client
{   volatile boolean menuItemSelected = false;

    public Client( JMenu topMenu )
    {   // Add an item to the topMenu, and arrange to be notified when it
        // is selected:

        JMenuItem myItem = new JMenuItem( "Hello" );
        myItem.addActionListener
        (   new ActionListener()
            {   public void actionPerformed( ActionEvent e )
                {   menuItemSelected = true;                // process selection
                }
            }
        );

        topMenu.add( myItem );
    }
}
```

This anonymous-inner-class version seemed, at first, pretty strange to me. Once I got used to the weird syntax, I came to prefer the anonymous-inner-class style because it lets me put all three parts of the Observer pattern (the publisher reference, the subscriber reference, and the activity to perform on publication) in one place. It's rather like a `for` statement, which lets you put all the parameters of loop control in one place.

Observer simplifies the code by passing the notification directly to the interested client, rather than through some visual intermediary.

Now let's apply the Observer pattern to Life. The `Clock` class, shown in Listing 3-1, uses Observer to notify a subscriber (the `Universe` object) of clock-tick events. The Clock has the role of Subject/Publisher. The Concrete Observer/Subscriber role is filled by any class that implements the `Listener` (Observer) interface (Listing 3-1, line 93). In this example, the Observer interface defines only one method (`void tick()`), but the pattern doesn't prohibit more complex interfaces.

The `Universe` object (the Concrete Observer/Subscriber) subscribes to the "tick" event like this:

```
Clock.instance().addClockListener
(   new Clock.Listener()
    {   public void tick()
        {   // code to handle a clock tick goes here
        }
    }
);
```

Clock is a Singleton, a reference to which is fetched by Clock.instance(). (The complete code from which the previous snippet is extracted is in Listing 3-7, which I'll discuss later in the chapter. If you want to skip ahead, the previous code is on line 140 of Listing 3-7, p. 142.)

The anonymous Clock.Listener derivative has the role of Concrete Observer/Subscriber. (In Figure 3-1, this anonymous Clock.Listener derivative is the one immediately to the left of the Clock class—the one that's connected to the Neighborhood indirectly via the Universe.)

It's actually arguable whether the Universe is the Concrete Observer or the anonymous inner class that actually receives the message is the Concrete Observer. Conceptually, it's the Universe, but physically, it's the inner-class object. In the UML, the inner-class-ness of the declaration is indicated by the circle with the plus in it, and the arrow indicates that messages are sent from the event handler object to the Universe object in the course of handling the tick.

As an aside, notice in the earlier code that Clock is a "classic" Gang-of-Four Singleton—only one instance of it exists, and it's accessed through a static accessor method that creates the instance (Clock.instance()). The private constructor (Listing 3-1, line 21) guarantees uniqueness, and the accessor method is declared on line 27. This Singleton can't be reified using the everything-is-static or the make-the-instance-reference-static mechanism because the clock constructor modifies the look of the menu bar, and the constructor cannot do that until the menu bar exists. When I tried to use one of the simpler reifications, I found that the Clock Singleton was being created too early, so the menu wasn't being set up properly. A "classic" Singleton solves the problem.

The Concrete Observer/Subscriber (the Universe instance) registers itself with the Publisher by calling addClockListener() (Listing 3-1, line 89), which delegates to an object of class Publisher, which we'll look at momentarily. The Universe object starts the clock by calling Clock.instance().startTicking(), and thereafter, the listener is notified at periodic intervals. (The tick() method of the registered listener is called.)

The tick management is handled by a java.util.Timer object declared on Listing 3-1, line 14. This is yet another example of Observer. The startTicking method on line 39 passes a scheduleAtFixedRate() message to a TimerTask object, whose run() method is called every time the timer "expires." This particular timer is set up to be recurrent, so it expires (and calls run()) at periodic intervals determined by the millisecondsBetweenTicks argument.

Listing 3-1. *Clock.java: The* Clock *Class*

```
 1  package com.holub.life;
 2
 3  import java.awt.*;
 4  import java.awt.event.*;
 5  import javax.swing.*;
 6  import java.util.*;
 7  import java.util.Timer;      // overrides java.awt.timer
 8  import com.holub.ui.MenuSite;
 9  import com.holub.tools.Publisher;
10
11  /**...*/
12
13  public class Clock
14  {   private Timer           clock       = new Timer();
```

```
    private TimerTask       tick        = null;

    // The clock can't be an everything-is-static Singleton because
    // it creates a menu, and it can't do that until the menus
    // are established.
    //
    private Clock()
    {   createMenus();
    }

    private static Clock instance;

    public synchronized static Clock instance()
    {   if( instance == null )
            instance = new Clock();
        return instance;
    }

    /** Start up the clock.
     *  @param millisecondsBetweenTicks The number of milliseconds between
     *                        ticks. A value of 0 indicates that
     *                        the clock should be stopped.
     */

    public void startTicking( int millisecondsBetweenTicks )
    {   if(tick != null)
        {   tick.cancel();
            tick=null;
        }

        if( millisecondsBetweenTicks > 0 )
        {   tick =  new TimerTask()
                    {   public void run(){ tick(); }
                    };
            clock.scheduleAtFixedRate( tick, 0, millisecondsBetweenTicks);
        }
    }

    public void stop()
    {   startTicking( 0 );
    }

    private void createMenus()
    {
        // First set up a single listener that will handle all the
        // menu-selection events except "Exit"

```

```
        ActionListener modifier =
            new ActionListener()
            {   public void actionPerformed(ActionEvent e)
                {
                    String name = ((JMenuItem)e.getSource()).getName();
                    char toDo = name.charAt(0);

                    if( toDo=='T' )
                        tick();                 // single tick
                    else
                        startTicking(   toDo=='A' ? 500:      // agonizing
                                        toDo=='S' ? 150:      // slow
                                        toDo=='M' ? 70 :      // medium
                                        toDo=='F' ? 30 : 0 ); // fast
                }
            };

        MenuSite.addLine(this,"Go","Halt",              modifier);
        MenuSite.addLine(this,"Go","Tick (Single Step)",modifier);
        MenuSite.addLine(this,"Go","Agonizing",         modifier);
        MenuSite.addLine(this,"Go","Slow",              modifier);
        MenuSite.addLine(this,"Go","Medium",            modifier);
        MenuSite.addLine(this,"Go","Fast",              modifier);
    }

    private Publisher publisher = new Publisher();

    public void addClockListener( Listener observer )
    {   publisher.subscribe(observer);
    }

    public interface Listener
    {   void tick();
    }

    public void tick()
    {   publisher.publish
        (   new Publisher.Distributor()
            {   public void deliverTo( Object subscriber )
                {   ((Listener)subscriber).tick();
                }
            }
        );
    }
}
```

Implementing Observer: The *Publisher* Class

It turns out that Observer can be surprisingly difficult to implement, particularly in an environment such as Swing, where several threads may interact.

Swing notifications, such as menu-selection events, are processed on an "event" thread that's created by the Swing subsystem. Many Swing applications are single threaded in that `main()` does nothing but create a few windows and then terminate. All actual processing is done on the Swing event thread in response to some user input action. Nonetheless, I've worked on several systems where the main object model was running on the main thread (among others) and creating Swing user-interface elements on the fly. Since the Swing notifications are issued from the Swing event thread, Swing sends lots of asynchronous messages to the main object model at unpredictable times. Since the Swing code on the event-handler thread and the code on the main thread can access the same objects simultaneously, a collision is unavoidable unless you synchronize properly.

The Swing event thread is not directly accessible to you, so unless you add or remove subscribers in event handlers (possible but unlikely), it's possible for the subscriber list to be modified on a user thread while notifications are in progress on the Swing event thread. Since both threads are accessing the same subscriber list, you're in trouble.

Unfortunately, a publisher implementation such as the following just won't work in this environment.

```
class Publisher1
{   ArrayList subscribers = new ArrayList();

    public synchronized void subscribe( Runnable subscriber )
    {   subscribers.add( subscriber );
    }

    public synchronized void cancelSubscription( Runnable subscriber )
    {   subscribers.remove( subscriber);
    }

    private synchronized void fireEvent()    // notify all subscribers
    {   for( int i = 0; i < subscribers.size(); ++i )
            ((Runnable) subscribers.get(i) ).run();
    }
}
```

It's reasonable that the subscriber list be modified during notification, and the notification cycle could take some time. You don't know how long it will take for `run()` to run, since that code is provided by the client class. Locking the `subscribe()` method during the entire notification period may "starve" the thread that's trying to subscribe because notifications could happen one after the other, and the subscribing thread may never be able to get in.

If you remove the synchronization from `fireEvent()` to eliminate the "starvation," then you introduce an equally nasty problem. The `fireEvent()` method can execute on one thread while the `subscribe()` or `cancel()` method executes on a different thread. Without synchronization, it's possible for the list to be accessed in the middle of a modification, corrupting the `subscribers` list as a consequence.

Turning the tables, again, there's something to be said in favor of synchronizing everything. In an unsynchronized situation, if the subscribers you add while notifications are in progress are tacked onto the end of the list, the subscriber can be notified of an event that happened before it subscribed! The event happens, notifications start and are preempted, the new subscriber is added, and then the subscriber is notified. The synchronized version of `fireEvent()` doesn't have this problem.

So what's a mother to do? You have several approaches. The first is to use the `Collection` interface rather than a concrete-class name (which I had to do earlier to be able to call `get()`) and use an `Iterator` to traverse the list.

```
class Publisher2
{   private Collection subscribers = new LinkedList();

    public synchronized void subscribe( Runnable subscriber )
    {   subscribers.add( subscriber );
    }

    public synchronized void cancelSubscription( Runnable subscriber )
    {   subscribers.remove( subscriber);
    }

    private void fireEvent()
    {   for( Iterator i = subscribers.iterator(); i.hasNext(); )
            ((Runnable) i.next() ).run();
    }
}
```

I'm leveraging the fact that `add(...)` and `remove(...)` throw an exception if they're called while an iterations in progress. Therefore, attempts to register a listener while notifications are going on will result in an exception toss, and the thread that attempted to add the listener will have to try again later. This solution is obviously not ideal: it dumps too much work on the shoulders of the calling object.

Another approach uses copying.

```
class Publisher3
{   private Collection subscribers = new LinkedList();

    public synchronized void subscribe( Runnable subscriber )
    {   subscribers.add( subscriber );
    }

    public synchronized void cancelSubscription( Runnable subscriber )
    {   subscribers.remove( subscriber);
    }

    private void fireEvent()
    {   Collection localCopy;
```

```
        synchronized( this )
        {   localCopy = localCopy.clone();
        }

        for( Iterator i = subscribers.iterator(); i.hasNext(); )
            ((Runnable) i.next() ).run();
    }
}
```

I've used `clone()` to make a copy of the subscriber list. (I must synchronize while copying.) Then I notify the subscribers from the copy. Since the original list isn't used during notification, I can now modify that list without impacting the notification process. This approach solves the problems I've been discussing, but it introduces a few new ones.

First, it's possible for the publisher to notify a subscriber after the subscriber has canceled its subscription (because notifications are being made from the copy). This problem exists in all the Swing observers and is a problem with Observer generally. In practice, observers are rarely removed, so this problem is probably not worth solving. If you know that a notification can arrive after removal, then you can write the code defensively.

The second copying-related problem is worth putting some effort into. It's just unacceptable to make a copy every time an event is fired, which can be frequently. It's better to make copies only when subscribers cancel their subscriptions, which happens rarely in practice.

The `Publisher` class (Listing 3-3, later) solves the too-much-copying problem elegantly (if I do say so myself). Listing 3-2 shows an excerpt from the `Clock` class that shows you how it handles Observer. The `addClockListener(...)` method just delegates to the `Publisher`. The `tick()` method, which is called every time the clock "ticks," notifies all the observers. It does this by passing a Command object that actually does the notification to the `Publisher`. That is, the `Publisher` calls the `Distributor()` derivative's `deliverTo()` method multiple times, passing it a different subscriber on the list each time.

Because the Command object encapsulates the knowledge of how to notify an Observer, the `Publisher` can delegate the mechanics of notification to the Command object. The `Publisher` doesn't need to know how to actually notify subscribers.

The subscriber-specific information is in the Command object, not the `Publisher`. The `Publisher` manages the list of subscribers, and it knows that `Distributor` derivatives know how to notify subscribers, so the `Publisher` can delegate the notification process to the `Distributor`. This way, the `Publisher` doesn't need to know anything about the `Clock.Listener` interface.

Listing 3-2. *Implementing Observer with a* `Publisher` *Object*

```
private Publisher publisher = new Publisher();

public interface Listener
{   void tick();
}

public void addClockListener( Listener l )
{   publisher.subscribe(l);
}

```

```
public void tick()
{   publisher.publish
    (   new Publisher.Distributor()
        {   public void deliverTo( Object subscriber )
            {   ((Listener)subscriber).tick();
            }
        }
    );
}
```

Turning to Listing 3-3, the `Publisher` object maintains a linked list of subscribers. (The head-of-list reference is declared on line 117.) I've implemented the linked list myself rather than using the `LinkedList` class, primarily because `LinkedList` doesn't support operations I need (appending a list segment to another list, for example). My original implementation was actually built around `LinkedList`, but the implementation was large, messy, and hard to maintain. A singly linked list is trivial to implement in any event, and I saw no point in writing bad code solely to support an existing data-structure class.

Each node in the list is represented by an instance of the `Node` class (Listing 3-3, line 92), which holds references to the subscriber and the next `Node` in the list. The constructor both creates a new node and links that node into the list, at its head. I pass the constructor references to both the new subscriber and the current head-of-list pointer. The node puts itself at the head of the list by initializing its `next` reference to the old head reference. The `subscribe()` method (Listing 3-3, line 133) sets the head-of-list reference to the newly created `Node` object. All the fields of `Node` are final, so the `Node` is an "immutable" object. It cannot change once it's created. Consequently, it's safe for multiple threads to access a given `Node` object simultaneously with no need for synchronization.

The top part of Figure 3-4 shows the message flow (I'll discuss the bottom part of this figure in the "Implementing Observer: The *Publisher* Class" section). When an event occurs, the client class calls the `publish()` method on line 128. The `publish()` method just traverses the list from head to tail, asking each subscriber to "accept" the `deliveryAgent` Command object that was passed as an argument to `publish(...)`.

Looking at the `Node`'s `accept(...)` method (Listing 3-3, line 112), you'll see that all `accept()` does is ask the `deliveryAgent` to actually do the work of notification (by calling `deliverTo(...)`). The `deliveryAgent` actually notifies the subscriber that the event occurred. By using a Command object to hide the notification mechanics, I move those mechanics out of the Publisher itself, making it much more flexible. I'll have more to say on this issue in the next section.

As a design aside, since the `Node` class is an inner class of `Publisher`, you could argue reasonably that I should dispense with the `accept()` method entirely and modify the `publish(...)` method to invoke `deliverTo` directly, as follows:

```
public void publish( Distributor deliveryAgent )
{   for(Node cursor = subscribers; cursor != null; cursor = cursor.next)
        deliveryAgent.deliverTo( cursor.subscriber );
}
```

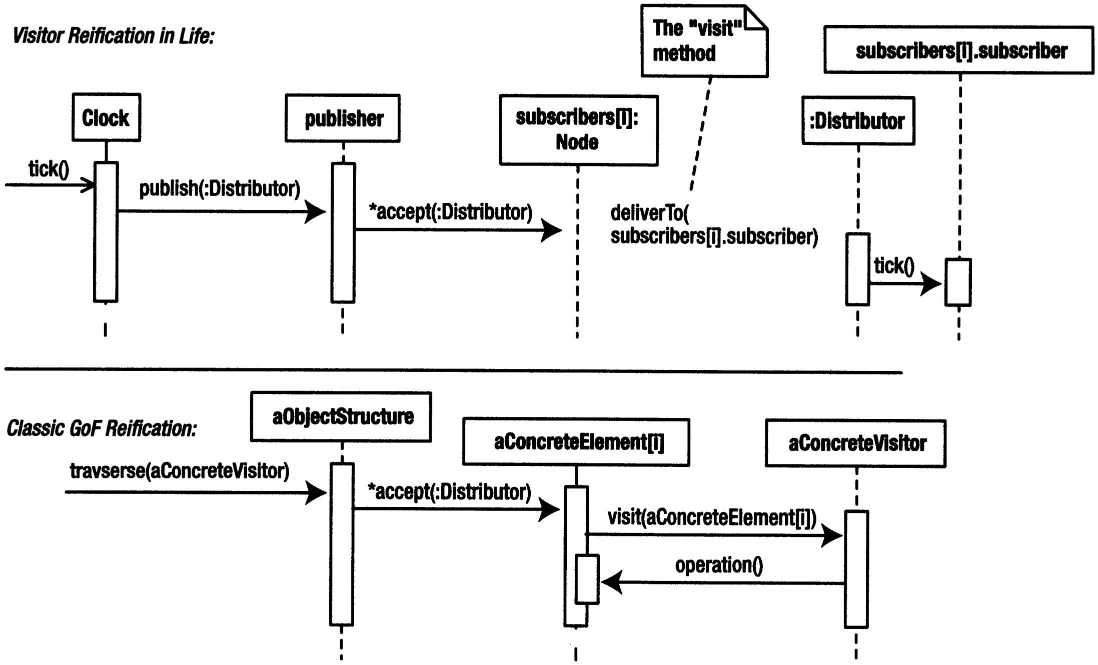


Figure 3-4. *The dynamic behavior of Visitor*

This change would simplify the code, but when I put on my designer hat, I don't like it that the `Publisher` object accesses a private field of `Node` (`subscriber`) as if it were `public`. Simply because Java permits this sort of back-door access of inner classes does not mean that it's a good thing—it strengthens the coupling relationships between the two classes unnecessarily. I like to treat inner classes such as `Node` as if they were top-level classes with respect to access. If you do let an outer-class method violate the inner-class's declared access privilege, at least do it with your eyes open, knowing that you're trading off a bit of maintainability to trivially simplify the code. (Sometimes—when the inner class is effectively a C-like `struct` with no methods—direct access is reasonable. I'll permit myself to do this only when the inner class is declared `private`, however.)

Getting back to the publication process, remember from a few paragraphs back that `Node` objects are immutable—they can't be modified after creation—and new `Node` objects are inserted at the head of the list. The ramifications of this add-to-head-of-list strategy are significant when notifications are in progress. Figure 3-5 shows the situation that occurs when one thread calls `subscribe(d)` just after another thread calls `publish(...)`. The list is being modified while notifications are in progress. The new node (in gray) was not in the list when publication begins, so the first subscriber to be notified is `c`. All subscribers to the right of `c` are also notified as the `Publisher` traverses the list, but `d` may as well not be in the list, at least for the purposes of this particular notification. It's perfectly safe, then, to add nodes to the list while publication is in progress, and I haven't had to copy the list (or synchronize anything) to achieve this safety.

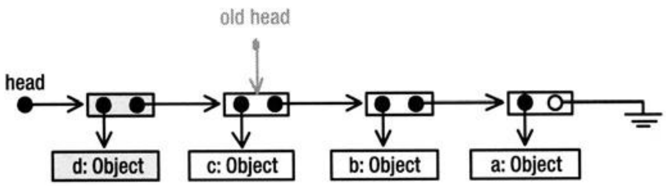


Figure 3-5. *Adding a subscriber while notifications are in progress*

The removal process is a bit more involved, primarily because I'm using a recursive algorithm. Many programmers seem to think recursive algorithms are inherently "bad" (opaque and inefficient), but in this case, the use is appropriate. Recursive algorithms are indeed hard to understand at times, but the inefficiency argument is often bogus. For example, because the list in the present code is singly linked, I would have to keep a stack of references to all the nodes that I have visited because once I find the node to delete, I'll need a list of that node's predecessors. Keeping this list is trivial in a recursive implementation—the list elements are just local variables in each recursive call. Doing the same thing manually with some sort of stack would use roughly the same memory as the recursive solution and make the code larger and more complex. I saw no point in using a mechanism that was more complex and no more efficient (at least in terms of space) than the recursive one simply to eliminate the recursion. I could also have solved the need-to-know-your-predecessor's-problem by making the list doubly linked, but that would also have added a bunch of unnecessary complexity. Figure 3-6 shows "before" and "after" pictures of the removal process. In the "before" image, I am removing node b (in gray). The bottom image shows what things look like after the remove. Interestingly, I have added two nodes but haven't actually removed anything. Any traversals that were in progress in the original list will continue as if nothing had happened, because nothing has happened to the original list. I've also moved the head pointer to the newly added far-left node. New traversals will begin at the new head-of-list reference, so they will not include the node I removed. Once any ongoing traversals complete, there will be no external references to any of the nodes in the dashed box, so they will all be garbage collected.

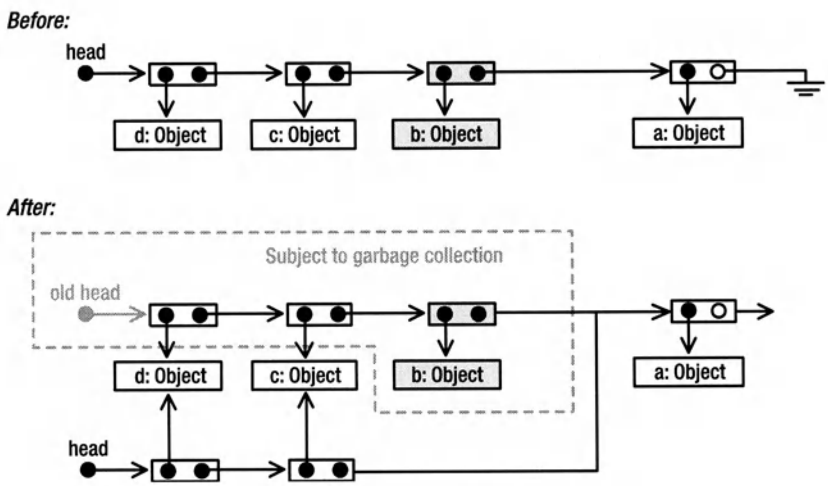


Figure 3-6. *Removing a subscriber while notifications are in progress*

Looking at the code, the recursion simplifies the code at the expense of clarity. The `cancelSubscription()` method (Listing 3-3, line 137) just delegates to the head-of-list node's `remove()` method (Listing 3-3, line 101). This recursive method first traverses down to the node to delete. It then starts returning back up to the original call. It does nothing with the node to delete, but as it returns it creates the new nodes for everything to the left of the deleted one and initializes the new nodes to point at the original subscribers.

Listing 3-3. *Publisher.java: A Subscription Manager*

```
1  package com.holub.tools;
2
3  /*******************************************************************
4   * This class replaces the Multicaster class that's described in
5   * <i>Taming Java Threads</i>. It's better in almost every way
6   * (It's smaller, simpler, faster, etc.). The primary difference
7   * between this class and the original is that I've based
8   * it on a linked-list, and I've used a Strategy object to
9   * define how to notify listeners, thereby making the interface
10  * much more flexible.
11  * <p>
12  * Publisher class provides an efficient thread-safe means of
13  * notifying listeners of an event. The list of listeners can be
14  * modified while notifications are in progress, but all listeners
15  * that are registered at the time the event occurs are notified (and
16  * only those listeners are notified). The ideas in this class are taken
17  * from the Java's AWTEventMulticaster class, but I use an (iterative)
18  * linked-list structure rather than a (recursive) tree-based structure
19  * for my implementation. The observers are notified in the opposite
20  * order that they were added.
21  * <p>
22  * Here's an example of how you might use it:
23  * <PRE>
24  *  class EventGenerator
25  *  {   interface Listener
26  *      {   notify( String why );
27  *      }
28  *
29  *      private Publisher publisher = new Publisher();
30  *
31  *      public void addEventListener( Listener l )
32  *      {   publisher.subscribe(l);
33  *      }
34  *
35  *      public void removeEventListener ( Listener l )
36  *      {   publisher.cancelSubscription(l);
37  *      }
38  *      public void someEventHasHappend(final String reason)
```

```
 *      {   publisher.publish
 *          (   new Publisher.Distributor()
 *              {   public void deliverTo( Object subscriber )
 *                  {   ((Listener)subscriber).notify(reason);
 *                  }
 *              }
 *          );
 *      }
 *  }
 * </PRE>
 * Since you're specifying what a notification looks like
 * by defining a Listener interface, and then also defining
 * the message passing symantics (inside the Distributor derivative),
 * you have complete control over what the interface looks like.

. . .
 */

public class Publisher
{
    public interface Distributor
    {   void deliverTo( Object subscriber );    // the Visitor pattern's
    }                                           // "visit" method.

    // The Node class is immutable. Once it's created, it can't
    // be modified. Immutable classes have the property that, in
    // a multithreaded system, access does not have to be
    // synchronized, because they're read-only.
    //
    // This particular class is really a struct so I'm allowing direct
    // access to the fields. Since it's private, I can play
    // fast and loose with the encapsulation without significantly
    // impacting the maintainability of the code.

    private class Node
    {   public final Object subscriber;
        public final Node   next;

        private Node( Object subscriber, Node next )
        {   this.subscriber = subscriber;
            this.next       = next;
        }

        public Node remove( Object target )
        {   if( target == subscriber )
                return next;

```

```
            if( next == null )                    // target is not in list
                throw new java.util.NoSuchElementException
                                                (target.toString());

            return new Node(subscriber, next.remove(target));
        }

        public  void accept( Distributor deliveryAgent ) // deliveryAgent is
        {   deliveryAgent.deliverTo( subscriber );      // a "visitor"
        }
    }

    private volatile Node subscribers = null;

    /** Publish an event using the deliveryAgent. Note that this
     *  method isn't synchronized (and doesn't have to be). Those
     *  subscribers that are on the list at the time the publish
     *  operation is initiated will be notified. (So, in theory,
     *  it's possible for an object that cancels its subscription
     *  to nonetheless be notified.) There's no universally "good"
     *  solution to this problem.
     */

    public void publish( Distributor deliveryAgent )
    {   for(Node cursor = subscribers; cursor != null; cursor = cursor.next)
            cursor.accept( deliveryAgent );
    }

    public void subscribe( Object subscriber )
    {   subscribers = new Node( subscriber, subscribers );
    }

    public void cancelSubscription( Object subscriber )
    {   subscribers = subscribers.remove( subscriber );
    }

    //------------------------------------------------------------------
    private static class Test
    {
        static final StringBuffer actualResults   = new StringBuffer();
        static final StringBuffer expectedResults = new StringBuffer();

        interface Observer
        {   void notify( String arg );
        }

        static class Notifier
```

```
    {   private Publisher publisher = new Publisher();

        public void addObserver( Observer l )
        {   publisher.subscribe(l);
        }

        public void removeObserver ( Observer l )
        {   publisher.cancelSubscription(l);
        }

        public void fire( final String arg )
        {   publisher.publish
            (   new Publisher.Distributor()
                {   public void deliverTo( Object subscriber )
                    {   ((Observer)subscriber).notify(arg);
                    }
                }
            );
        }
    }

    public static void main( String[] args )
    {
        Notifier source = new Notifier();
        int      errors = 0;

        Observer listener1 =
            new Observer()
            {   public void notify( String arg )
                {   actualResults.append( "1[" + arg + "]" );
                }
            };

        Observer listener2 =
            new Observer()
            {   public void notify( String arg )
                {   actualResults.append( "2[" + arg + "]" );
                }
            };

        source.addObserver( listener1 );
        source.addObserver( listener2 );

        source.fire("a");
        source.fire("b");

        expectedResults.append("2[a]");
```

```
            expectedResults.append("1[a]");
            expectedResults.append("2[b]");
            expectedResults.append("1[b]");

            source.removeObserver( listener1 );

            try
            {   source.removeObserver(listener1);
                System.err.print("Removed nonexistent node!");
                ++errors;
            }
            catch( java.util.NoSuchElementException e )
            {   // should throw an exception, which we'll catch
                // (and ignore) here.
            }

            expectedResults.append("2[c]");
            source.fire("c");

            if( !expectedResults.toString().equals(actualResults.toString()) )
            {
                System.err.print("add/remove/fire failure.\n");
                System.err.print("Expected:[");
                System.err.print( expectedResults.toString() );
                System.err.print("]\nActual:  [");
                System.err.print( actualResults.toString() );
                System.err.print("]");
                ++errors;
            }

            source.removeObserver( listener2 );
            source.fire("Hello World");
            try
            {   source.removeObserver( listener2 );
                System.err.println("Undetected illegal removal.");
                ++errors;
            }
            catch( Exception e ) { /*everything's okay, do nothing*/ }

            if( errors == 0 )
                System.err.println("com.holub.tools.Publisher: OKAY");
            System.exit( errors );
        }
    }
}
```

The Clock Subsystem: The Visitor Pattern

Now let's refocus on the `Publisher` from the design-pattern perspective. The notion of passing to every node of some data structure a Command object that either uses or modifies that node is immortalized in the **Visitor** pattern.

Here are the roles that the various Life classes take on within the pattern:

- **`Clock`** is the Client.
- **`Publisher`** is the Object Structure.
- **`Distributor`** is the Visitor interface.
- **`Node`** is the contained Element.
- **`Node.accept()`** is the "accept" request.
- **`deliverTo()`** is the "visit" request.
- An anonymous **`Distributor`** derivative created by **`Clock`** is the Concrete Visitor.

Figure 3-4, which you looked at earlier, shows the UML for both the actual reification and the "classic" Gang-of-Four reification of Visitor. In the `Publisher` reification at the top of the Figure, an external "client" (the `Clock`) does something to or with the objects in some container (the `Publisher`) by passing a Visitor object (a class that implements `Distributor`) to that container. The container handles the traversal, asking each object to "accept" the visitor. The object then turns around and passes a "visit" method to each visitor, passing it an interface to itself or equivalent.

This reification has only one difference between it and the "classic" Gang-of-Four reification: The Visitor object in the "classic" version is passed a reference to the accepting Element, and the visitor then turns around and performs some operation on that Element. In other words, the visitor is passed a reference to the Element that it must access. Otherwise, the Visitor wouldn't know how to send messages to the Element. (If you can remember back that far, the Car-and-Map example in Chapter 1 uses the same strategy. In fact, a car is a Visitor to the Road Element.) In the `Publisher` reification, however, the `Node` Element exposes one of its fields (the `subscriber` reference) to the `Distributor` Visitor by passing it as an argument to the Visitor's `deliverTo(...)` method.

One other reification of Visitor needs mentioning. Instead of passing the Visitor a reference to the Element itself or passing the Visitor one or more fields of Element, you can pass the Visitor a reference to an interface that provides restricted access to the Element. In other words, the interface in the Element role can represent a subset of the interface to the actual object. This way the Element object can tightly control what the Visitor can do to it.

Another common extension involves heterogeneous lists. The `Publisher` class's Visitor interface (`Distributor`) is trivial, having only one method. If the Object Structure is heterogeneous, it's reasonable for the Visitor interface to implement several versions of the "visit" request, one for each Element type. It's also reasonable for the Element (the `Node`) to implement some interface so that the traversal code can be decoupled from the actual element type. For example:

```
interface Visitor
{   public visit( NodeType           aNode );
    public visit( AnotherNodeType    aNode );
    public visit( YetAnotherNodeType aNode );
}
```

This way the Concrete Visitor doesn't have to guess which possible node type it's dealing with.

Whew! That's pretty complicated. Fortunately, Visitor is as hard as it gets. Visitor is one of the most complicated—and hardest to understand—Gang-of-Four patterns, but it's pretty useful when applied correctly. Now that you, I hope, understand the mechanics, let's look at why I used Visitor at all.

Java has a class called `AWTEventMulticaster` that works a lot like the `Publisher` class. Using it, you can make a list of literally any listener that's supported by AWT. Here's how you'd implement a list of `ActionListener` objects:

```
public myComponent extends Component
{
    ActionListener subscribers = null;

    public synchronized void addActionListener(ActionListener subscriber)
    {   subscribers = AWTEventMulticaster.add(subscribers, subscriber);
    }

    public synchronized void removeActionListener(ActionListener subscriber)
    {   subscribers = AWTEventMulticaster.remove(subscribers, subscriber);
    }

    public void fire()
    {   if (subscribers != null)
            subscribers.actionPerformed(new ActionEvent(/*...*/));
    }
}
```

Here's how you'd implement a list of TextListener objects:

```
public myComponent extends Component
{
    TextListener subscribers = null;

    public synchronized void addTextListener(TextListener subscriber)
    {   subscribers = AWTEventMulticaster.add(subscribers, subscriber);
    }

    public synchronized void removeTextListener(TextListener subscriber)
    {   subscribers = AWTEventMulticaster.remove(subscribers, subscriber);
    }
```

```
    public void fire()
    {   if (subscribers != null)
            subscribers.textValueChanged(new TextEvent(/*...*/));
    }
}
```

The chameleon-like adaptability comes from AWTEventMulticaster implementing literally every listener interface supported by AWT and of course, implementing all the methods of every listener interface. That's a lot of work, it's hard to maintain, and the class carries around the baggage of implementing dozens of methods, only one of which is typically used in a given application.

The general problem that AWTEventMulticaster is trying to solve is how to implement a generalized event publisher where the various event handlers take arbitrary arguments and return arbitrary values. AWTEventMulticaster solves the problem by implementing all the event handlers that the designers could imagine, but that's a lot of work and requires modification of the original class if you need to add a handler that you didn't imagine.

Returning your thoughts to the Publisher, I am solving the same problem as the designers of AWTEventMulticaster. I want to be able to publish arbitrary events to arbitrary subscribers. I could apply the same kitchen-sink mentality to the Publisher by supporting a few generic interfaces. Here's one possibility that can handle three types of subscribers, each of which can handle a different number of arguments in the event-notification method:

```
class BruteForcePublisher
{
    Node head = null;

    interface Subscriber0
    {   public void eventFired();
    }
    interface Subscriber1
    {   public void eventFired(Object arg1);
    }
    interface Subscriber2
    {   public void eventFired(Object arg1, Object arg2);
    }

    class Node
    {   //...
        private Object  subscriber;
        private Node    next;

        public void fire()
        {   ((Subscriber0)subscriber).eventFired();
        }

        public void fire(Object arg1 )
        {   ((Subscriber1)subscriber).eventFired(arg1);
        }
```

```
        public void fire(Object arg1, Object arg2)
        {   ((Subscriber2)subscriber).eventFired(arg1,arg2);
        }
    }

    public void fire()
    {   for( Node current = head; current != null; current = current.next )
            current.fire();
    }

    public void fire(Object arg1)
    {   for( Node current = head; current != null; current = current.next )
            current.fire(arg1);
    }

    public void fire(Object arg1, Object arg2)
    {   for( Node current = head; current != null; current = current.next )
            current.fire(arg1, arg2);
    }
}
```

Even if you can stomach that all the arguments have to be declared as `Object` so can't be type checked, this solution has a lot of problems. What if I want to add a subscriber whose interface requires two methods? I'd have to add the following to my class definition:

```
class BruteForcePublisherV2
{
    //...

    interface Subscriber1x2
    {   public void event1Fired(Object arg1, Object arg2);
        public void event2Fired(Object arg1, Object arg2);
    }

    class Node
    {   //...
        public void fire2(  Object e1Arg1, Object e1Arg2,
                            Object e2Arg1, Object e2Arg2)
        {   ((Subscriber2)subscriber).event1Fired(e1Arg1, e1Arg2);
            ((Subscriber2)subscriber).event2Fired(e2Arg1, e2Arg2);
        }
    }

    //...

    public void fire( Object e1Arg1, Object e1Arg2,
                      Object e2Arg1, Object e2Arg2 )
```

```
    {   for( Node current = head; current != null; current = current.next )
            current.fire(e1Arg1, e1Arg2, e2Arg1, e2Arg2);
    }
}
```

In fact, every time I need to add another event type, I need to add a new interface and two methods to the class. This is way too much work.

To the rescue comes the Visitor pattern. The basic idea is that you often traverse collections of objects passing messages to the object that comprise the collection. In the current example, I'm traversing a list of subscribers, passing event notifications to each subscriber. The problem with this naive implementation is that I need to add a method to the data-structure element (the `Node`) every time I add a new event type to the system, but the odds of calling that particular method in a given chunk of code is small.

Visitor solves the problem with a Command object. The idea is to pass the Node a Command object (the Visitor) that understands how to notify a particular kind of listener. This way the `Node` doesn't have to support every possible listener type. You saw this process earlier in Listing 3-2.

The main downside of Visitor, other than its obvious complexity, is that the Visitor is external to the Node but nonetheless can modify or otherwise accesses what would normally be `private` components of the Node. This violation of encapsulation flies in the face of one of the central precepts of OO systems: data abstraction. A Visitor can access the Element strictly through a public interface, however, and I strongly recommend you do that whenever possible.

It's also difficult to maintain Visitor-based systems because changes to the Nodes require parallel changes in the Visitor interface and all its derivatives. It's exactly this rippling effect of a change that OO systems are designed to avoid. Use Visitor only when the interface is expected to be stable.

The Menuing Subsystem: Composite

Now let's move to the menuing subsystem in the lower-left corner of Figures 3-1 and 3-2 on pages 84 and 85.

The first pattern of interest in the menuing system is **Composite**. I'll explain how Composite is used now. You'll see how it's implemented in Life later.

Composite simplifies the management of a hierarchy of similar objects by letting a container treat everything that it contains identically, even if the contained objects are actually instances of different classes. If you look at a containment hierarchy as a kind of tree, the containers are the interior nodes.

Composite is used extensively in the current Life implementation, but Java's menuing system provides a scaled-down example, so let's start there. Figure 3-7 and Figure 3-8 show the two menus that my Life implementation supports, and Figure 3-9 shows the containment hierarchy for these menus. (Figure 3-9 also shows—in gray at the bottom—how you can hook a submenu into the system.)

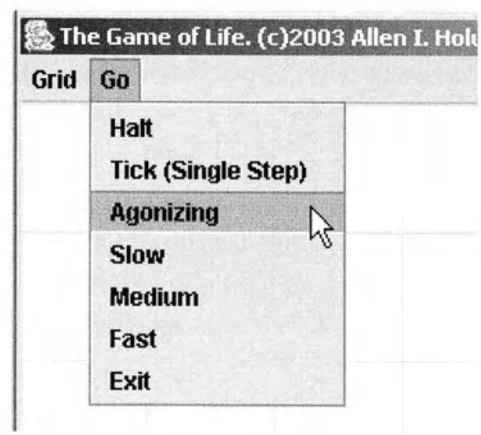


Figure 3-7. *Life's Go menu*

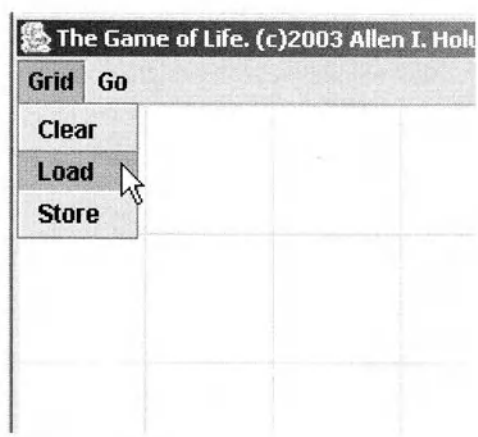


Figure 3-8. *Life's Grid menu*

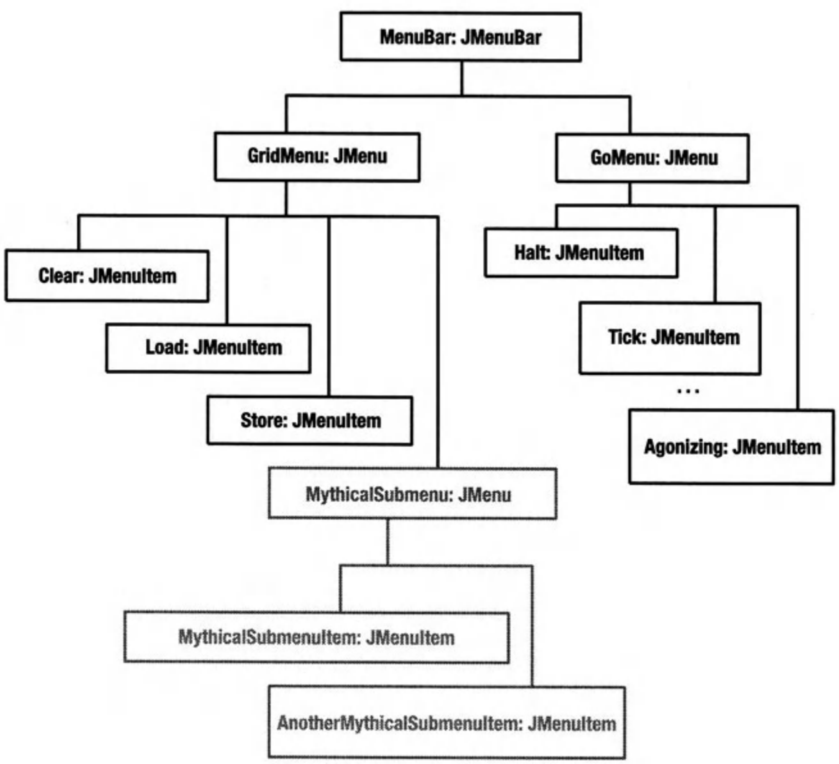


Figure 3-9. *The menu containment hierarchy*

Here are the characteristics of Composite:

- An object hierarchy is split up into two main classes of objects, both of which typically implement the same interface.
- The common interface serves the role of Component.
- One or more of the classes of objects in the hierarchy serve in the role of Leaf—objects of these classes form the leaves (terminate the branches) of the object tree. They don't contain anything.
- Another of the classes of objects in the hierarchy serve in the role of Composite—objects of these classes contain objects that implement the same interface as do the Composite objects.
- When you write the code for the Composite, you don't need to know whether a contained object is a Leaf or another Composite because you can access them through the interface they both implement. The code is much simpler to write as a consequence.

The following translates the general description to the menuing system:

- The menu hierarchy consists of `JMenuItem` and `JMenu` objects.
- The `JMenuItem` serves in the role of Leaf.
- The `JMenu` serves in the role of Composite. `JMenu` extends `JMenuItem`, so a `JMenu` is a `JMenuItem`.
- The programmers who wrote the code to handle menus don't need to know whether a menu item is a Leaf (a `JMenuItem`) or a Composite (a `JMenu` representing a submenu) because the two can be treated identically. That is, a menu can contain both `JMenu` and `JMenuItem` objects, but the `JMenu` can be treated as a `JMenuItem`.

This example differs from the "classic" Gang-of-Four example in that `JMenuItem` serves in two roles. It acts simultaneously as the Component interface and a Leaf node. In a "classic" reification, `JMenuItem` and `JMenu` would implement a common interface, and `JMenu` would not extend `JMenuItem`. Neither architecture is superior to the other; both are legitimate reifications of the pattern—two different ways to accomplish the same objective.

You'll notice that the AWT `Component`/`Container` hierarchy also satisfies these requirements so reifies Composite. Figure 3-10 shows the UML. A `Container` is a `Component`, as are all the leaf nodes, such as `Button`, that don't contain anything. A container can lay out its subcomponents without regard to their actual class, since all the subcomponents implement the (effective) `Component` interface.

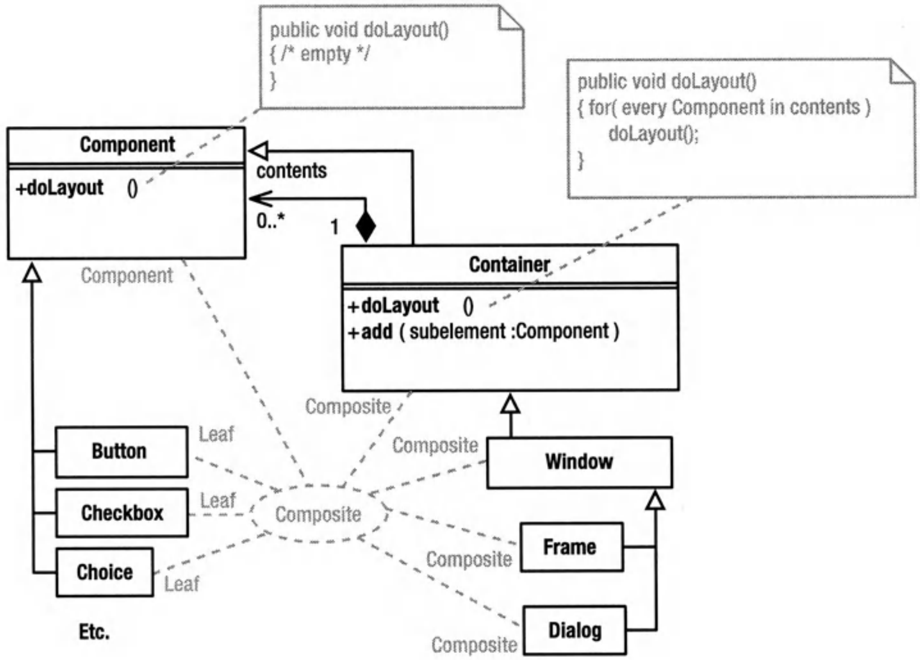
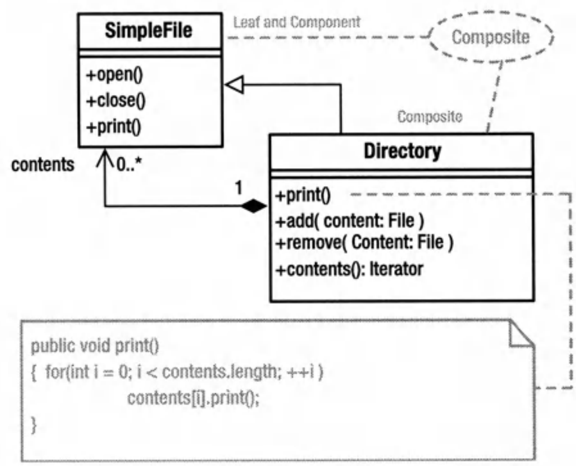


Figure 3-10. *The AWT component-container system*

A directory system is another a natural example of Composite. (A *directory* is a file that contains other files, including subdirectories. In Unix/Linux systems, a directory is literally a file, in fact. You can open it, read its contents, and so on.



In this simple example, the `SimpleFile` class serves in both the Component and the Leaf roles. (In the "classic" Gang-of-Four example, the classes in the Leaf and Composite roles both implement a common Component interface.)

One common source of confusion with Composite is really obvious in the UML for a Directory at the bottom of the previous page. The structure of the object hierarchy inverts the class-hierarchy structure. When drawing the object hierarchy, a root node is a container, typically shown at the top of the tree, with the leaf nodes below it. In composite, however, this root node is a subclass, typically shown beneath the leaf-node class in the UML diagrams.

The important characteristic of the pattern is that when you're traversing a directory system, you don't need to know whether the subdirectories are files or subdirectories. The code shown in the comment, previously, just sends a `print()` message to the object. If it's a `SimpleFile`, then the single filename is printed. If it's a `Directory`, then its contents are printed. Because both sorts of components can be treated uniformly, the methods of the class in the Composite role are easy to write.

I'll come back to Composite in the context of Life in a bit, but let's continue exploring the menuing subsystem. Listing 3-4 demonstrates how you'd have to build the menuing system using the raw APIs. A lot of things can wrong with this code, the most obvious of which is that it's way too long. None of the code is particularly complicated, but there's a lot of it. Moreover, building a menu is a repetitive task, and when you build menus by hand all over the place, you have lots of repetitive code. `ProtoUniverse.addMenus(...)` and `ProtoClock.addMenus(...)` are almost identical. I'm also not happy with the clutter. I really don't want to be worrying about the details of the menu APIs when I'm working on "business" classes (classes that implement key design abstractions).

A more serious problem, from a design point of view, is that a `ProtoUniverse` is what's called a *key abstraction* of the design (a "business" object). Its characteristics are determined by the problem definition, and it's part of the user's mental model of the problem. ProtoClock and `Neighborhood` are also key abstractions. To say that a `Neigborhood` is a Frame or a Menu Contributor is nonsensical. You don't talk about your next-door "framers" (unless they're in the construction trade). You talk about your neighbors. Similarly, you don't say you live in a nice Menu Contributor; you live in a nice neighborhood. For derivation to make sense in a design, the subclass must be the same thing as the superclass, though it might behave a little differently.

The basic drill for adding a menu item is as follows:

```
JMenuItem item = new JMenuItem( "Visible Text" );
item.setName("someInternalName");
item.addActionListener( handlerToCallWhenItemSelected )
containingMenu.add( item );
```

The "name" you establish with `setName` is an arbitrary string that's stored internally in the `JComponent`. (All `JComponents` have one.) The name is not visible to the user at all. (The visible label—called the *text*—is either passed into the constructor, as shown here, or set with a call to `setText()`). The point of an internal "name" is that the visible text could change over time, but the internal name won't. I use the internal name, not the visible label, when I decide which menu item was selected (in the `switch` statement on line 65 of Listing 3-4, for example).

One thorny problem emerges when you look at the code in Listing 3-4 really closely. That cast on line 20 is ugly. I want to be able to create a UI at this level using the Composite pattern. A `JFrame` is a `JComponent` that holds other `JComponents`: I want to be able to treat all subwindows as `JComponent` objects. I need that cast to exercise the `MenuContributor` abilities of the `ProtoUniverse`, however. You should avoid casts generally—they're a source of runtime errors. Were `JComponent` an interface rather than a class, I could neatly solve the problem by changing the `ProtoUniverse` declaration to read as follows:

```
class ProtoUniverse extends JPanel
            implements Cell, JComponent, MenuContributor
```

But `JComponent` isn't an interface, so I'm stuck. I can't change the source code for AWT and Swing. This difficulty demonstrates why it's a good idea to use interfaces from Day One.

Another "solution" to the cast problem also doesn't work: Make `MenuContributor` an abstract class that extends `JComponent` to add a few methods, and then define all my references as references to this new class. The `ProtoClock` contributes to the menu but doesn't display a UI, however. I don't want a `ProtoClock` to carry around the literal and metaphorical baggage of a `JComponent`, so deriving it from `JComponent` is inappropriate.

Short of a major refactor of AWT/Swing, no ideal solution exists to this problem, so I'll just let the cast stand for now.

Listing 3-4. *Building a Menuing System with the Raw APIs*

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

interface MenuContributor
{   void addMenus( JMenuBar menuBar );
}

class Menus extends JFrame
{
    public Menus()
    {

        JComponent      theUniverse    = new ProtoUniverse();
        MenuContributor theClock       = new ProtoClock();
        //...

        JMenuBar menuBar = new JMenuBar();
        theClock.                      addMenus( menuBar );
        ((MenuContributor)theUniverse).addMenus( menuBar );

        JMenuItem exit = new JMenuItem("Exit");
        exit.addActionListener
        (   new ActionListener()
            {   public void actionPerformed(ActionEvent e)
                {   System.exit(0);
                }
            }
```

```
        );
        menuBar.add(exit);

        menuBar.setVisible(true);
        setJMenuBar( menuBar );

        getContentPane().add( theUniverse );
        setDefaultCloseOperation( EXIT_ON_CLOSE );
        pack();
        setSize( 200, 200 );
        show();
    }

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

class ProtoClock implements MenuContributor
{   //...

    public void addMenus( JMenuBar menuBar )
    {
        JMenuItem halt = new JMenuItem("Halt");
        JMenuItem slow = new JMenuItem("Slow");
        JMenuItem fast = new JMenuItem("Fast");
        //...

        halt.setName( "halt" );
        slow.setName( "slow" );
        fast.setName( "fast" );

        ActionListener handler =
            new ActionListener()
            {   public void actionPerformed(ActionEvent e)44
                {   String name = ((JMenuItem)e.getSource()).getName();
                    switch( name.charAt(0) )
                    {
                    case 'h':   setClockSpeed( 0 ); break;
                    case 'f':   setClockSpeed( 500 );   break;
                    case 's':   setClockSpeed( 250 );   break;
                    }
                }
            };

        halt.addActionListener( handler );
        slow.addActionListener( handler );
```

```
        fast.addActionListener( handler );

        JMenu go = new JMenu( "Go" );
        go.add( halt );
        go.add( slow );
        go.add( fast );

        menuBar.add( go );
    }

    private void setClockSpeed( int speed )
    {   System.out.println( "Changing speed to " + speed );
    }
}

class ProtoUniverse extends JPanel implements Cell, MenuContributor
{   //...
    public void addMenus( JMenuBar menuBar )
    {
        JMenuItem clear = new JMenuItem("Clear");
        JMenuItem load  = new JMenuItem("Load");
        JMenuItem store = new JMenuItem("Store");
        //...

        clear.setName( "clear" );
        load.setName( "load" );
        store.setName( "store" );

        ActionListener handler =
            new ActionListener()
            {   public void actionPerformed(ActionEvent e)
                {   String name = ((JMenuItem)e.getSource()).getName();
                    switch( name.charAt(0) )
                    {
                    case 'c':   clearGrid();    break;
                    case 'l':   loadGrid(); break;
                    case 's':   storeGrid();    break;
                    }
                }
            };

        clear.addActionListener( handler );
        load.addActionListener ( handler );
        store.addActionListener( handler );

        JMenu grid = new JMenu( "Grid" );
        grid.add( clear );
```

```
        grid.add( load );
        grid.add( store );

        menuBar.add( grid );

    }

    // stubs:

    private void clearGrid(){ System.out.println("clear"); }
    private void loadGrid() { System.out.println("load");   }
    private void storeGrid(){ System.out.println("store"); }
}

interface Cell
{   //...
}

```

The Menuing Subsystem: Facade and Bridge

Now that you know how the underlying menuing system works, you're ready to look at the actual code (MenuSite.java in Listing 3-6, on page 123).

The `MenuSite` is an example of the **Facade** design pattern. The point of Facade is to make it easier to access a complex system via a simple one. The main problem with the raw menuing APIs I just discussed is that it's just too complicated to build a menuing system. You need to create lots of classes, nest them together properly, and hook up listeners. All this work does nothing but clutter up the code unnecessarily and make the code hard to maintain. The main point of the `MenuSite` Facade is to hide this complexity and let you build a menu with a few simple method calls.

This particular Facade also nicely solves a few OO-design issues. You'll remember (I hope) from Chapter 1 that it's best for objects to create their own user interfaces so as not to expose implementation information to a UI-builder object. Put another way, a screen in an object-oriented user interface is typically an aggregate of smaller user interfaces that individual objects in the system provide. This way, when you change an object's structure, you also change the way that it presents itself. If both the business and presentation logic are in the same class definition, then the scope of your change is limited to that class definition. You don't have to go out and find all the screen-builder classes and change them too.

Implementing this aggregate-UI structure can be vexing with any menuing system, which are typically treated procedurally as a monolithic object. It's reasonable, however, for an object to want to add menu items that are related to itself to the main menu bar. For example, Life's `Clock` class needs to add a menu that handles changes in clock speed. Nothing else in the system is particularly interested in that menu, so the `Clock` should create (and install) it.[1]

---

1. This user-interface architecture, by the way, is not Model/View/Controller. It's called Presentation/Action/Control and is discussed in the book *A System of Patterns: Pattern-Oriented Software Architecture* by Buschmann, et. al. (John Wiley & Sons, 1996).

A great example of this sort of object structure is in Microsoft's Object-Linking-and-Embedding In-Place-Activation system. (I'm not particularly happy with the way that Microsoft implemented their architecture, but the concepts are solid.) When you want to put a numeric table into a Microsoft Word document, you select Insert:Object:Microsoft Excel Worksheet from the main menu. Word launches Excel, and the two programs negotiate how to share a common user interface. Word gives Excel a portion of the screen to work with (into which Excel puts its spreadsheet UI) and Excel puts menu items onto the main Word menu bar. While you're working on the spreadsheet, you're actually talking to the "Excel" object through a user interface created by that object. Excel also pops up various toolbars and other UI elements.

When you click outside Excel's window, the Excel object shuts down. Excel removes all its pop-up windows, removes from the menu bar any items that it added, and returns to Word an image to display in place of the Excel-generated user interface. Finally, Excel returns a "blob" of data to Word—a byte array that Word keeps for Excel until the next activation.

This "blob" of data is an example of the **Memento** pattern, which I'll discuss in greater depth later in this chapter. The data blob is the Memento, Excel is the Originator, and Word is the Caretaker. The point of Memento is that the Caretaker (Word) has no idea what's in the Memento—Word certainly can't manipulate the stored spreadsheet. The Caretaker (Word) just holds onto the Memento until the Originator (Excel) needs it again.

Returning to the UI issues, the Excel object creates its own user interface but integrates this UI into an existing framework UI managed by Word. OLE activation is a great example of object-oriented structure. Some high-level object owns the application's main frame and menu bar, but the actual UI is an aggregation of smaller user interfaces contributed by various objects in the system. The point of this structure is to isolate the objects' implementations from the rest of the system. When Microsoft comes out with a new version of Excel that has new UI requirements, it does not have to modify Word at all. The UI changes are concentrated in Excel itself. The same reasoning applies to smaller objects; when you need to make a structural change to the object that impacts the UI (such as adding a new clock speed), all the changes are concentrated in a single class definition (the `Clock`), and you don't need to change anything in the encapsulating program.

The *MenuSite*

The current Life implementation solves the menuing problem by using a class (`MenuSite`) that allows you to approach the menuing system in an object-oriented way. The point of the `MenuSite` class is twofold: to simplify the interface to the menuing system and to make it easy for objects to contribute to a shared menu bar. The notion of talking to an entire subsystem (or at least a group of related classes) through a single simple interface is embodied in the **Facade** design pattern. Facade provides a simple way to perform some task that would otherwise be complicated. It's perfectly reasonable to make a Facade that simplifies one aspect of what a subsystem does, but elsewhere in your code, you'd talk to the subsystem directly, without using the Facade. A Facade doesn't necessarily isolate you from changes to subsystem. Nonetheless, a Facade can provide this isolation if you're careful to access subsystem classes through only the

Facade. The Bridge design pattern (discussed in a subsequent chapter) can force subsystem isolation by prohibiting direct access to subsystem classes. That is, a Facade provides assistance with a subsystem while a Bridge isolates you from that subsystem completely.

Since the `MenuSite` interface is key, let's look at how to use it. You must first "bind" it to a top-level frame window. The `Life` class (Listing 3-5) does it on line 26 with the following method call:

```
MenuSite.establish( this );
```

The `Life` object does only two things: It creates the main frame and installs the `MenuSite` into it, and (on line 30) it creates the game board (the `Universe`) and installs it in the frame. It's often the case that an OO system's `main()` does nothing but create a few high-level objects, hook them together, and terminate. Remember, an OO system is a network of cooperating objects. There's no spider in the middle of the web pulling the strands. Put another way, there's no "god" class that controls the workings of the program.

Listing 3-5. *Life.java*

```
 1  package com.holub.life;
 2
 3  import java.awt.*;
 4  import javax.swing.*;
 5  import com.holub.ui.MenuSite;
 6
 7  /*******************************************************************
 8   * An implementation of Conway's Game of Life.
 9   * @author Allen I. Holub
10   */
11
12  public final class Life extends JFrame
13  {
14      private static JComponent universe;
15
16      public static void main( String[] arguments )
17      {   new Life();
18      }
19
20      private Life()
21      {   super( "The Game of Life. "
22                  +"&copy;2003 Allen I. Holub <http://www.holub.com>");
23
24          // Must establish the MenuSite very early in case
25          // a subcomponent puts menus on it.
26          MenuSite.establish( this );
27
28          setDefaultCloseOperation    ( EXIT_ON_CLOSE         );
29          getContentPane().setLayout  ( new BorderLayout()    );
30          getContentPane().add( Universe.instance(), BorderLayout.CENTER);
```

```

        pack();
        setVisible( true );
    }
}
```

The MenuSite object is an everything-is-static Singleton. That means you can have only one menu bar in a program. I thought about allowing multiple menu bars, but the problem of finding a particular menu site turned out to be pretty complicated, so I took the easy way out. I'm willing to concede the point if you think my decision was too limiting.

Once the site is established, any object can add or remove menus by calling static methods. The Clock class's createMenus method (Listing 3-1, line 57, on page 91) sets up the menus for the Clock object to use. The method starts by creating a single ActionListener object—a Concrete Observer that is shared by most of the line items on the menu. This particular observer starts up a java.util.Timer object at the speed indicated by the selected item:

```
ActionListener modifier =
    new ActionListener()
    {   public void actionPerformed(ActionEvent e)
        {
            String name = ((JMenuItem)e.getSource()).getName();
            char toDo = name.charAt(0);

            if( toDo=='T' )
                tick();                     // single tick
            else
                startTicking(   toDo=='A' ? 500:      // agonizing
                                toDo=='S' ? 150:      // slow
                                toDo=='M' ? 70 :      // medium
                                toDo=='F' ? 30 : 0 ); // fast
        }
    };
```

The method then sets up the menus by calling MenuSite.addLine(...) several times (reproduced next). The first argument identifies the object that "owns" the menu item. The second argument specifies the menu to which the item is added. In this case, it's added to the Go menu on the main menu bar. Since no Go menu exists, The MenuSite automatically creates the new Menu and places it on the menu bar. The third argument is the "name" of this particular line item. (Menu items, like all Components, have both an invisible "name" and a visible "text" attribute. By default, the MenuSite uses the same string for both purposes.) The final argument is the Observer to notify when a user selects this menu item.

```
MenuSite.addLine(this,"Go","Halt",                modifier);
MenuSite.addLine(this,"Go","Tick (Single Step)",modifier);
MenuSite.addLine(this,"Go","Agonizing",           modifier);
MenuSite.addLine(this,"Go","Slow",                modifier);
MenuSite.addLine(this,"Go","Medium",              modifier);
MenuSite.addLine(this,"Go","Fast",                modifier);
```

It's not done here, but if the `Clock` object wanted to remove all the menu items that it added, it could call this:

```
MenuSite.removeMyMenus(this);
```

Similarly, the `Clock` can disable all the menu items it adds by calling this:

```
MenuSite.setEnable(this, false);
```

The point of this structure, again, is that it makes it easy for a particular object to manage only those menu items that it's interested in and for the rest of the system to not care about how a given object is using the menu. Table 3-1 shows the remainder of the documentation for `MenuSite`.

Listing 3-6 shows the entire source code for `MenuSite`. The implementation of this Facade doesn't have any design patterns, so I won't spend any time on it. (This isn't a book on GUI building, after all.) The main point of including the entire listing in this book is to demonstrate how much complexity the Facade is hiding. This complex mess would be in the midst of your code were the Facade not there. You'll find several pages of code that demonstrates how to use a `MenuSite` in the `MenuSite.Test.main(...)` method, starting on line 548 of Listing 3-6. If you want to skip the listing, turn to page 139.

Table 3-1. `MenuSite` *Documentation*

`public static void establish(JFrame container)`: Establishes a `JFrame` as the program's menu site. This method must be called before any of the other menu-site methods may be called. (Most of these will throw a `NullPointerException` if you try.)

`public static void addMenu(Object requester, String menuSpecifier)`: Creates and adds an empty menu to the menu bar. Menus are generally created by `addLine(...)`. This method is provided for situations where one requester creates a menu structure and other requesters add line items to this structure. The requesters that added the line items can remove those items without removing the menu that contained the items.

Menus are inserted on the menu bar just to the left of the Help menu. (The "help" menu [a menu whose name is the string `help`—case is ignored] is special in that it always appears on the far right of the menu bar.) Use `addLine(...)` to add line items to the menu. This method does the name-to-label substitution described in `addLine(...)`as well. As in `addLine(...)`, the name string also defines the (visible) label if no mapping is found.

If the requested menu already exists, this method silently does nothing.

Parameters:

`requester`: The object that "owns" this menu. All menus (and line items) added by a specific requester are removed by a single `removeMyMenus(...)` call. The requester need not be the actual object that adds the menu—there may not be a single one—it is simply used to identify a group of menu items that will be removed in bulk. All items that have the same requester object are removed at once.

`menuSpecifier:` The menu to create. A simple specifier (with no colons in it) creates an item on the menu bar itself. Submenus are specified using the syntax `main:sub`. For example, the following call creates a New submenu under the File menu:

```
addMenu( this, "File:New" )
```

If the supermenu (in this example, File) doesn't exist, it's created. You can have more than one colon if you want to go down more than one level (for example, `Edit:Text:Size`). Up to six levels below the menu bar (six colons) are supported. (If you have more than that, you should seriously reconsider your menu structure.) Intermediate menus are added as necessary.

`public static void addLine(Object requester, String toThisMenu, String name, ActionListener observer)`: Adds a line item to a menu. The menu is created if it does not already exist. This method is the preferred way to both create menus and add line items to existing menus. See `addMenu(...)` for the rules of menu creation.

The name parameter is used for both the name and visible text, but you can specify text different from the name by calling `addMapping(...)` (which can also be used to define shortcuts).

Parameters:

`requester`: The object that requested that this line item be added.

`name`: The (hidden) name text for this item. When there's no name map (see `addMapping(...)`), the same string is used for both the name and the label; otherwise the name argument specifies the name only, and the associated label (and shortcut) is taken from the map.

Use the name `"-"` to place a separator into a menu. The listener argument is not used in this case and can be null.

`toThisMenu`: The specifier of the menu to which you're adding the line item. (See `addMenu(...)` for a discussion of specifiers.) The specified menu is created if it doesn't already exist.

`listener`: The `ActionListener` to notify when the menu item is selected.

`public static void removeMyMenus(Object requester)`: Removes all items that were added by this requester. For the time being, the case of "foreign" items being placed on a menu created by another requester is not handled. Consider a program in which two objects both add an item to the File menu. The first object to add an item will be the official "owner" of the menu, since it created the menu. When you call `removeMyMenus()` for this first object, you want to remove the line item it added to the File menu, but you don't want to remove the File menu itself because it's not empty. Right now, the only solution to this problem is for a third requester to create the menu itself using `addMenu(...)`

---

continues

`public static void setEnable(Object requester, boolean enable)`: Disables or enables all menus and menu items added by a specific requester. You can disable a single menu item by using this:

```
MenuSite.getMyMenuItem(requester,"parent:spec","name").setEnabled(false);
```

Parameters:

> **`enable`**: Set this to true to enable all the requester's menu items.

`public static JMenuItem getMyMenuItem(Object requester, String menuSpecifier, String name)`: Gets a menu item for external modification.

Parameters:

> **`requester`**: The object that inserted the menu or item.
>
> **`menuSpecifier`**: The menu specifier passed to the original `addMenu(...)` or `addLine(...)` call.
>
> **`name`**: The name passed to `addLine(...)`; should be `null` if you want a menu rather than a line item within the menu.

Returns:

> The underlying `JMenu` or `JMenuItem`. Returns `null` if the item doesn't exist.

`public static void mapNames(URL table) throws IOException`: Establishes a "map" of (hidden) names to (visible) labels and shortcuts. Establishing a map changes the behavior of `addLine(...)` and `addMenu(...)` in that the specified ("text") label and shortcut are installed automatically for all names specified in the table. A map must be specified before the item named in the map are added to the menu site. You may call this method multiple times to load multiple maps, but the "name" component of each entry must be unique across all maps.

Parameters:

> **`table`**: A `Properties`-style file that maps named keys to labels, along with an optional shortcut. The general form is as follows:
>
> ```
> name1 = label One; C
> name2 = label Two; Alt X
> ```
>
> You can specify the shortcut in one of two ways. If it's a single character, as in the first example, the platform-default modifier is used. For example, in the first example, the shortcut will be a Ctrl+C in Windows, Command+C on the Mac, and so on. Otherwise, the shortcut specifier must take the form described in `javax.swing.KeyStroke.getKeyStroke(String)`. For example:
>
> - F1
> - control DELETE
> - typed a
> - alt shift released X
> - alt shift X

Names such as DELETE and F1 are shorthand for VK_DELETE and VK_F1. (You can find the complete set of VK_X constants in the java.awt.event.KeyEvent class.) You can use any of these "virtual" keys simply by removing the VK_.

F10 is hard-mapped to display the main menu (so that you can navigate the menus with the arrow keys, I assume). You could probably defeat this behavior with a key binding, but it's easier to just accept it as a fait accompli and not try to define F10 as a keyboard shortcut.

The input file is a standard Properties file, which is assumed to be ISO 8859-1 (not Unicode) encoded. ASCII works just fine, but see Properties.load(java.io.InputStream) for a full description of the file format.

---

public static void addMapping(String name, String label, String shortcut): Adds a name-to-label mapping manually. A mapping must be specified before the item is added to the menu site.

Parameters:

name: The menu-item name passed to addMenu(...) or addLine(...).

label: The visible label for that item.

shortcut: The shortcut, if any. Should be an empty string ("") if no shortcut is required. See mapNames(java.net.URL) for information on how to form this string.

Listing 3-6. *MenuSite.java*

```
package com.holub.ui;

import java.io.*;
import java.util.*;
import java.util.logging.*;
import java.util.regex.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**...*/

public final class MenuSite
{
    private static JFrame       menuFrame   = null;
    private static JMenuBar     menuBar     = null;

    /**...*/
    private static Map requesters = new HashMap();

    /**...*/
```

```
    private static Properties nameMap;

    /**...*/
    private static Pattern shortcutExtractor =
                Pattern.compile(
                        "\\s*([^;]+?)\\s*"                // value
                        +"(;\\s*([^\\s].*?))?\\s*$" );   // ; shortcut

    /**...*/
    private static Pattern submenuExtractor =
                Pattern.compile( "(.*?)(?::(.*?))?"
                                    + "(?::(.*?))?"
                                    + "(?::(.*?))?"
                                    + "(?::(.*?))?"
                                    + "(?::(.*?))?"
                                    + "(?::(.*?))?" );

    /**...*/

    private static final LinkedList menuBarContents =
                                                new LinkedList();

    /**...*/
    private MenuSite()

    /**...*/

    private static boolean valid()
    {   assert menuFrame != null : "MenuSite not established";
        assert menuBar   != null : "MenuSite not established";
        return true;
    }

    /**...*/
    public synchronized static void establish(JFrame container)
    {
        assert container != null;
        assert menuFrame == null:
                            "Tried to establish more than one MenuSite";

        menuFrame = container;
        menuFrame.setJMenuBar( menuBar = new JMenuBar() );

        assert valid();
    }

    /**...*/
```

```
    public static void addMenu( Object requester, String menuSpecifier )
    {   createSubmenuByName( requester, menuSpecifier );
    }

    /**...*/
    public static void addLine( Object requester,
                                String toThisMenu,
                                String name,
                                ActionListener listener)
    {
        assert requester  != null: "null requester" ;
        assert name       != null: "null item"      ;
        assert toThisMenu != null: "null toThisMenu";
        assert valid();

        // The "element" field is here only so that we don't create
        // a menu if the assertion in the else clause fires.
        // Otherwise, we could just create the items in the
        // if and else clauses.

        Component element;

        if( name.equals("-") )
            element = new JSeparator();
        else
        {   assert listener != null: "null listener";

            JMenuItem lineItem = new JMenuItem(name);
            lineItem.setName( name );
            lineItem.addActionListener( listener );
            setLabelAndShortcut( lineItem );

            element = lineItem;
        }

        JMenu found = createSubmenuByName( requester, toThisMenu );
        if( found==null )
            throw new IllegalArgumentException(
                        "addLine() can't find menu ("+ toThisMenu +")" );

        Item item = new Item(element, found, toThisMenu );
        menusAddedBy(requester).add( item );
        item.attachYourselfToYourParent();
    }

    /**...*/

```

```
    public static void removeMyMenus( Object requester )
    {
        assert requester != null;
        assert valid();

        Collection allItems=(Collection)( requesters.remove(requester) );

        if( allItems != null )
        {   Iterator i = allItems.iterator();
            while( i.hasNext() )
            {   Item current = (Item) i.next();
                current.detachYourselfFromYourParent();
            }
        }
    }

    /**...*/
    public static void setEnable(Object requester, boolean enable)
    {
        assert requester != null;
        assert valid();

        Collection allItems = (Collection)( requesters.get(requester) );

        if( allItems != null )
        {
            Iterator i = allItems.iterator();
            while( i.hasNext() )
            {   Item current = (Item) i.next();
                current.setEnableAttribute(enable);
            }
        }
    }

    /**...*/

    public static JMenuItem getMyMenuItem(Object requester,
                                String menuSpecifier, String name)
    {
        assert requester        != null;
        assert menuSpecifier    != null;
        assert valid();

        Collection allItems = (Collection)( requesters.get(requester) );

        if( allItems != null )
        {   Iterator i = allItems.iterator();
```

```
            while( i.hasNext() )
            {   Item current = (Item) i.next();
                if( current.specifiedBy( menuSpecifier ) )
                {   if( current.item() instanceof JSeparator )
                        continue;

                    if( name==null && current.item() instanceof JMenu )
                        return (JMenu)( current.item() );

                    if(((JMenuItem)current.item()).getName().equals(name))
                        return (JMenuItem) current.item();
                }
            }
        }
        return null;
    }


    //============================================================
    //          Private support methods and classes            |
    //============================================================

    /**...*/
    private static JMenu createSubmenuByName( Object requester,
                                              String menuSpecifier )
    {
        assert requester != null;
        assert menuSpecifier != null;
        assert valid();

        Matcher m = submenuExtractor.matcher(menuSpecifier);
        if( !m.matches() )
            throw new IllegalArgumentException(
                                    "Malformed menu specifier.");

        // If it's null, then start the search at the menu bar;
        // otherwise start the search at the menu addressed by "parent"

        JMenuItem   child  = null;
        MenuElement parent = menuBar;
        String      childName;

        for(int i=1; (childName = m.group(i++)) != null; parent=child )
        {
            child = getSubmenuByName(childName,parent.getSubElements());

            if( child != null )
```

```
            {   if( !(child instanceof JMenu) ) // it's a line item!
                    throw new IllegalArgumentException(
                            "Specifier identifes line item, not menu.");
            }
            else // it doesn't exist, create it
            {
                child = new JMenu       (childName);
                child.setName           (childName );
                setLabelAndShortcut (child );

                Item item = new Item(child, parent, menuSpecifier );
                menusAddedBy(requester).add(item);
                item.attachYourselfToYourParent();
            }
        }

        return (JMenu)child; // the earlier instanceof guarantees safety
    }

    /**...*/

    private static JMenuItem getSubmenuByName( String name,
                                        MenuElement[] contents )
    {
        JMenuItem found = null;
        for( int i = 0; found==null && i < contents.length ; ++i )
        {
            // This is not documented, but the system creates internal
            // pop-up menus for empty submenus. If we come across one of
            // these, then look for "name" in the pop-up's contents. This
            // would be a lot easier if PopupMenu and JMenuItem
            // implemented a common interface, but they don't.
            // I can't use a class adapter to make them appear to
            // implement a common interface because the JPopupWindows
            // are manufactured by Swing, not by me.

            if( contents[i] instanceof JPopupMenu )
                found = getSubmenuByName( name,
                            ((JPopupMenu)contents[i]).getSubElements());

            else if( ((JMenuItem) contents[i]).getName().equals(name) )
                found = (JMenuItem) contents[i];
        }
        return found;
    }

    /**...*/
```

```java

public static void mapNames(URL table) throws IOException
{   if( nameMap == null )
        nameMap = new Properties();
    nameMap.load( table.openStream() );
}

/**...*/

public static void addMapping( String name, String label,
                                                    String shortcut)
{   if( nameMap == null )
        nameMap = new Properties();
    nameMap.put( name, label + ";" + shortcut );
}

/**...*/
private static void setLabelAndShortcut( JMenuItem item )
{   String name = item.getName();
    if( name == null )
        return;

    String label;
    if( nameMap != null
            && (label= (String)(nameMap.get(name))) != null )
    {
        Matcher m = shortcutExtractor.matcher(label);
        if( !m.matches() )  // Malformed input line
        {
            item.setText( name );
            Logger.getLogger("com.holub.ui").warning
            (
                "Bad "
                +"name-to-label map entry:"
                + "\n\tinput=[" + name + "=" + label + "]"
                + "\n\tSetting label to " + name
            );
        }
        else
        {   item.setText( m.group(1) );

            String shortcut = m.group(3);

            if( shortcut != null )
            {   if( shortcut.length() == 1 )
                {   item.setAccelerator
                    (   KeyStroke.getKeyStroke
```

```
                      ( shortcut.toUpperCase().charAt(0),
                        Toolkit.getDefaultToolkit().
                                        getMenuShortcutKeyMask(),
                        false
                      )
                    );
                }
                else
                {   KeyStroke key=KeyStroke.getKeyStroke(shortcut);
                    if( key != null )
                        item.setAccelerator( key );
                    else
                    {   Logger.getLogger("com.holub.ui").warning
                        ( "Malformed shortcut parent specification "
                            + "in MenuSite map file: "
                            + shortcut
                        );
                    }
                }
            }
        }
    }
}

/**...*/
private static Collection menusAddedBy( Object requester )
{
    assert requester  != null: "Bad argument"  ;
    assert requesters != null: "No requesters" ;
    assert valid();

    Collection menus = (Collection)( requesters.get(requester) );
    if( menus == null )
    {   menus = new LinkedList();
        requesters.put( requester, menus );
    }
    return menus;
}

/**...*/
private static final class Item
{
    // private JMenuItem  item;
    private Component  item;

    private String      parentSpecification; // of JMenu or of
                                             // JMenuItem's parent
```

```
        private MenuElement parent;              // JMenu or JMenuBar
        private boolean     isHelpMenu;

        public String toString()
        {   StringBuffer b = new StringBuffer(parentSpecification);
            if( item instanceof JMenuItem )
            {   JMenuItem i = (JMenuItem)item;
                b.append(":");
                b.append(i.getName());
                b.append(" (");
                b.append(i.getText());
                b.append(")");
            }
            return b.toString();
        }

        /*------------------------------------------------------------*/

        private boolean valid()
        {   assert item     != null : "item is null" ;
            assert parent   != null : "parent is null" ;
            return true;
        }

        /**...*/

        public Item( Component item, MenuElement parent,
                                    String parentSpecification )
        {   assert parent != null;
            assert parent instanceof JMenu || parent instanceof JMenuBar
                                : "Parent must be JMenu or JMenuBar";

            this.item          = item;
            this.parent        = parent;
            this.parentSpecification = parentSpecification;
            this.isHelpMenu  =
                    ( item instanceof JMenuItem )
                 && ( item.getName().compareToIgnoreCase("help")==0 );

            assert valid();
        }

        public boolean specifiedBy( String specifier )
        {   return parentSpecification.equals( specifier );
        }

        public Component item()
```

```
    {   return item;
    }

    /**...*/

    public final void attachYourselfToYourParent()
    {   assert valid();

        if( parent instanceof JMenu )
        {   ((JMenu)parent).add( item );
        }
        else if( menuBarContents.size() <= 0 )
        {   menuBarContents.add( this );
            ((JMenuBar)parent).add( item );
        }
        else
        {   Item last = (Item)(menuBarContents.getLast());
            if( !last.isHelpMenu )
            {
                menuBarContents.addLast(this);
                ((JMenuBar)parent).add( item );
            }
            else    // remove the help menu, add the new
            {       // item, then put the help menu back
                    // (following the new item).

                menuBarContents.removeLast();
                menuBarContents.add( this );
                menuBarContents.add( last );

                if( parent == menuBar )
                    parent = regenerateMenuBar();
            }
        }
    }

    /**...*/
    public void detachYourselfFromYourParent()
    {   assert valid();

        if( parent instanceof JMenu )
        {   ((JMenu)parent).remove( item );
        }
        else // the parent's the menu bar.
        {
            menuBar.remove( item );
            menuBarContents.remove( this );
```

```
                regenerateMenuBar(); // without me on it

                parent = null;
            }
        }

        /**...*/

        public void setEnableAttribute( boolean on )
        {   if( item instanceof JMenuItem )
            {   JMenuItem item = (JMenuItem) this.item;
                item.setEnabled( on );
            }
        }

        /**...*/
        private JMenuBar regenerateMenuBar()
        {   assert valid();

            // Create the new menu bar and populate it from
            // the current content's list.

            menuBar = new JMenuBar();
            ListIterator i = menuBarContents.listIterator(0);
            while( i.hasNext() )
                menuBar.add( ((Item)(i.next())).item );

            // Replace the old menu bar with the new one.
            // Calling setVisible causes the menu bar to be
            // redrawn with a minimum amount of flicker. Without
            // it, the redraw doesn't happen at all.

            menuFrame.setJMenuBar( menuBar );
            menuFrame.setVisible( true );
            return menuBar;
        }
    }

    /**...*/

    private static class Debug
    {
        public interface Visitor
        {   public void visit(JMenu e,int depth);
        }

        private static int traversalDepth = -1;
```

```

        /**...*/

        public static void visitPostorder( MenuElement me, Visitor v )
        {
            // If it's actually a JMenuItem (as compared to a
            // JMenuItem derivative such as a JMenu), then it's
            // a leaf node and has no children.

            if( me.getClass() != JMenuItem.class )
            {   MenuElement[] contents = me.getSubElements();
                for( int i=0; i < contents.length; ++i )
                {
                    if( contents[i].getClass() != JMenuItem.class )
                    {   ++traversalDepth;
                        visitPostorder( contents[i], v );
                        if( !(contents[i] instanceof JPopupMenu) )
                            v.visit((JMenu)contents[i], traversalDepth);
                        --traversalDepth;
                    }

                }
            }
        }
    }

    /**...*/
    public static class Test extends JFrame
    {
        static Test instance; // = new Test();
        static boolean isDisabled1 = false;
        static boolean isDisabled2 = false;

        Test()
        {
            setSize( 400, 200 );
            addWindowListener
            (   new WindowAdapter()
                {   public void windowClosing( WindowEvent e )
                    {   System.exit(1);
                    }
                }
            );
            MenuSite.establish( this );
            show();
        }

```

```
    //----------------------------------------------------------
    static class RemoveListener implements ActionListener
    {   public void actionPerformed( ActionEvent e )
        {   MenuSite.removeMyMenus( instance );
        }
    }
    //----------------------------------------------------------

    static public void main( String[] args ) throws Exception
    {
        com.holub.tools.Log.toScreen("com.holub.ui");
        UIManager.setLookAndFeel(
            UIManager.getSystemLookAndFeelClassName() );

        instance = new Test();

        // Create a generic reporter.

        ActionListener reportIt =
                new ActionListener()
                {   public void actionPerformed(ActionEvent e)
                    {   JMenuItem item = (JMenuItem)(e.getSource());
                        System.out.println( item.getText() );
                    }
                };


        // Create the File menu first.

        ActionListener terminator =
            new ActionListener()
            {   public void actionPerformed( ActionEvent e )
                {   System.exit(0);
                }
            };

        // Make the file menu with its own ID so that the removal
        // test in the main menu doesn't remove it.

        Object fileId = new Object();
        MenuSite.addMenu(fileId, "File" );
        MenuSite.addLine(fileId, "File", "Quit", terminator);
        MenuSite.addLine(fileId, "File", "Bye",  terminator);

        // Now, make a few more menus.

        MenuSite.addMenu(instance, "Main" );
```

```
        MenuSite.addLine
        (   instance, "Main", "Add Line Item to Menu",
            new ActionListener()
            {   public void actionPerformed( ActionEvent e )
                {   MenuSite.addLine(instance, "Main",
                        "Remove Main and Help menus",
                        new ActionListener()
                        { public void actionPerformed(ActionEvent e)
                          {  MenuSite.removeMyMenus(instance);
                          }
                        }
                    );
                }
            }
        );

        //------------------------------------------------------------
        MenuSite.addLine( instance, "Main", "-", null );
        //------------------------------------------------------------
        final Object disable1 = new Object();

        MenuSite.addLine(   instance, "Main", "Toggle1",
            new ActionListener()
            {   public void actionPerformed( ActionEvent e )
                {   isDisabled1 = !isDisabled1;
                    MenuSite.setEnable( disable1, !isDisabled1 );
                    MenuSite.getMyMenuItem(instance,
                                           "Main", "Toggle1").
                        setText
                        (   isDisabled1 ? "Enable following Item"
                                        : "Disable following Item"
                        );

                }
            }
        );
        MenuSite.getMyMenuItem(instance, "Main", "Toggle1").
                                setText("Disable following Item");

        MenuSite.addLine(disable1, "Main", "Disableable", reportIt);

        // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        final Object disable2 = new Object();

        MenuSite.addLine(   instance, "Main", "Toggle2",
            new ActionListener()
            {   public void actionPerformed( ActionEvent e )
```

```
                {   isDisabled2 = !isDisabled2;
                    MenuSite.setEnable( disable2, !isDisabled2 );
                    MenuSite.getMyMenuItem(instance,
                                            "Main", "Toggle2").
                        setText
                        (   isDisabled2 ? "Enable following Item"
                                        : "Disable following Item"
                        );
                }
            }
        );
        MenuSite.getMyMenuItem(instance, "Main", "Toggle2").
                                setText("Disable following Item");
        MenuSite.addLine(disable2, "Main", "Disableable", reportIt);

        //-----------------------------------------------------

        // Check that a single line item can be removed

        final Object id = new Object();

        MenuSite.addLine( id, "Main", "-", null );
        MenuSite.addLine
        (   id, "Main", "Remove this item & separator line",
            new ActionListener()
            {   public void actionPerformed( ActionEvent e )
                {   MenuSite.removeMyMenus( id );
                }
            }
        );

        // Check out submenus. Create two of them, one in two
        // steps and the other in a single step. Then add items
        // that remove the submenus to make sure that removal works
        // correctly.

        MenuSite.addLine(instance,"Main", "-", null );
        MenuSite.addLine(instance,
                "Main:Submenu1", "Submenu One Item", reportIt );
        MenuSite.addLine(instance,
                "Main:Submenu2", "Submenu Two Item", reportIt );
        MenuSite.addLine(instance,
                "Main:Submenu3", "Submenu Three Item", reportIt );
        MenuSite.addLine(instance,
                "Main:Submenu2:SubSubmenu2",
                "Sub-Submenu Two Item", reportIt );

```

```
            MenuSite.addLine(instance,
                    "Main:Submenu3:SubSubmenu3",
                    "Sub-Submenu Three Item", reportIt );

            MenuSite.addLine(instance,
                    "Main:Submenu3:SubSubmenu3:SubSubSubmenu3",
                    "Sub-Sub-Submenu Three Item", reportIt );

            MenuSite.addLine(instance, "Main", "-", null );

            // Check that the map file works correctly.
            // Items 5 and 6 are deliberately malformed in the map
            // file and will cause an error to be logged.
            // item.7 doesn't exist in the file.

            MenuSite.mapNames(
               new URL("file://c:/src/com/holub/ui/test/menu.map.txt"));

            MenuSite.addLine( instance, "Main", "item.1", reportIt );
            MenuSite.addLine( instance, "Main", "item.2", reportIt );
            MenuSite.addLine( instance, "Main", "item.3", reportIt );
            MenuSite.addLine( instance, "Main", "item.4", reportIt );
            MenuSite.addLine( instance, "Main", "item.5", reportIt );
            MenuSite.addLine( instance, "Main", "item.6", reportIt );
            MenuSite.addLine( instance, "Main", "item.7", reportIt );

            // Create a help menu. Do it in the middle of things
            // to make sure that it ends up on the far right.
            // Use all three mechanisms for adding menu items directly
            // using the menu's "name," and using the menu's "text").

            MenuSite.addLine( instance, "Help", "Get Help", reportIt );

            // Create a second "requester" and have it add a Removal
            // menu with the name RemovalMenu. Picking that menu
            // will remove only the menu for the current requester.
            // Do this after doing the help menu to make sure that
            // it's inserted in the right place.

            final Object x = new Object();
            MenuSite.addLine
            (   x,
                "Removal", "Select to Remove Removal menu",
                new ActionListener()
                {   public void actionPerformed(ActionEvent e)
                    {   MenuSite.removeMyMenus(x);
                    }
                }
```

```
                );
            }
        }
    }

```

The Core Classes

This section contains four listings that I'll be presenting in depth in the next few sections. The classes in these listings all participate in the same set of patterns, so it's best to put them together in one place. I don't expect you to read them now, however. Bookmark the subsections and refer to them later, then skip ahead to page 161.

This section really shows you one of the significant disadvantages of a hard-core design-pattern approach. My implementation of Life is probably the most complicated implementation of life ever written—way too complicated, given what it does. ("If it's that complicated, it must be wrong!") If you go nuts with the patterns and lose track of what you're actually trying to accomplish, you can introduce so much complexity into the code as to render it almost useless.

My goal in writing this code was as much to demonstrate design patterns as it was to build an optimal Life implementation, however. The SQL interpreter in the next chapter does not have this problem—it is production code.

The *Universe* Class

Listing 3-7 shows Universe.java.

Listing 3-7. *Universe.java*

```
package com.holub.life;

import java.io.*;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

import com.holub.io.Files;
import com.holub.ui.MenuSite;

import com.holub.life.Cell;
import com.holub.life.Storable;
import com.holub.life.Clock;
import com.holub.life.Neighborhood;
import com.holub.life.Resident;

/**
 * The Universe is a mediator that sits between the Swing
```

```
 * event model and the Life classes. It is also a Singleton,
 * accessed via Universe.instance(). It handles all
 * Swing events and translates them into requests to the
 * outermost Neighborhood. It also creates the Composite
 * Neighborhood.
 */

public class Universe extends JPanel
{   private         final Cell      outermostCell;
    private static  final Universe  theInstance = new Universe();

    /** The default height and width of a Neighborhood in cells.
     *  If it's too big, you'll run too slowly because
     *  you have to update the entire block as a unit, so there's more
     *  to do. If it's too small, you have too many blocks to check.
     *  I've found that 8 is a good compromise.
     */
    private static final int  DEFAULT_GRID_SIZE = 8;

    /** The size of the smallest "atomic" cell&mdash;a Resident object.
     *  This size is extrinsic to a Resident (It's passed into the
     *  Resident's "draw yourself" method.
     */
    private static final int  DEFAULT_CELL_SIZE = 8;

    // The constructor is private so that the universe can be created
    // only by an outer-class method [Neighborhood.createUniverse()].

    private Universe()
    {   // Create the nested Cells that comprise the "universe." A bug
        // in the current implementation causes the program to fail
        // miserably if the overall size of the grid is too big to fit
        // on the screen.

        outermostCell = new Neighborhood
                        (   DEFAULT_GRID_SIZE,
                            new Neighborhood
                            (   DEFAULT_GRID_SIZE,
                                new Resident()
                            )
                        );

        final Dimension PREFERRED_SIZE =
                        new Dimension
                        (  outermostCell.widthInCells() * DEFAULT_CELL_SIZE,
                           outermostCell.widthInCells() * DEFAULT_CELL_SIZE
                        );

```

```
        addComponentListener
        (   new ComponentAdapter()
            {   public void componentResized(ComponentEvent e)
                {
                    // Make sure that the cells fit evenly into the
                    // total grid size so that each cell will be the
                    // same size. For example, in a 64x64 grid, the
                    // total size must be an even multiple of 63.

                    Rectangle bounds = getBounds();
                    bounds.height /= outermostCell.widthInCells();
                    bounds.height *= outermostCell.widthInCells();
                    bounds.width  =  bounds.height;
                    setBounds( bounds );
                }
            }
        );

        setBackground   ( Color.white    );
        setPreferredSize( PREFERRED_SIZE );
        setMaximumSize  ( PREFERRED_SIZE );
        setMinimumSize  ( PREFERRED_SIZE );
        setOpaque       ( true           );

        addMouseListener
        (   new MouseAdapter()
            {   public void mousePressed(MouseEvent e)
                {   Rectangle bounds = getBounds();
                    bounds.x = 0;
                    bounds.y = 0;
                    outermostCell.userClicked(e.getPoint(),bounds);
                    repaint();
                }
            }
        );

        MenuSite.addLine( this, "Grid", "Clear",
            new ActionListener()
            {   public void actionPerformed(ActionEvent e)
                {   outermostCell.clear();
                    repaint();
                }
            }
        );

        MenuSite.addLine
        (   this, "Grid", "Load",
```

```
                new ActionListener()
                {   public void actionPerformed(ActionEvent e)
                    {   doLoad();
                    }
                }
            );

            MenuSite.addLine
            (   this, "Grid", "Store",
                new ActionListener()
                {   public void actionPerformed(ActionEvent e)
                    {   doStore();
                    }
                }
            );

            MenuSite.addLine
            (   this, "Grid", "Exit",
                new ActionListener()
                {   public void actionPerformed(ActionEvent e)
                    {   System.exit(0);
                    }
                }
            );

            Clock.instance().addClockListener
            (   new Clock.Listener()
                {   public void tick()
                    {   if( outermostCell.figureNextState
                                ( Cell.DUMMY,Cell.DUMMY,Cell.DUMMY,Cell.DUMMY,
                                  Cell.DUMMY,Cell.DUMMY,Cell.DUMMY,Cell.DUMMY
                                )
                          )
                        {   if( outermostCell.transition() )
                                refreshNow();
                        }
                    }
                }
            );
        }

        /** Singleton Accessor. The Universe object itself is manufactured
         *  in Neighborhood.createUniverse()
         */

        public static Universe instance()
        {   return theInstance;
```

```
    }

    private void doLoad()
    {   try
        {
            FileInputStream in = new FileInputStream(
               Files.userSelected(".",".life","Life File","Load"));

            Clock.instance().stop();        // stop the game and
            outermostCell.clear();          // clear the board.

            Storable memento = outermostCell.createMemento();
            memento.load( in );
            outermostCell.transfer( memento, new Point(0,0), Cell.LOAD );

            in.close();
        }
        catch( IOException theException )
        {   JOptionPane.showMessageDialog( null, "Read Failed!",
                    "The Game of Life", JOptionPane.ERROR_MESSAGE);
        }
        repaint();
    }

    private void doStore()
    {   try
        {
            FileOutputStream out = new FileOutputStream(
                  Files.userSelected(".",".life","Life File","Write"));

            Clock.instance().stop();        // stop the game

            Storable memento = outermostCell.createMemento();
            outermostCell.transfer( memento, new Point(0,0), Cell.STORE );
            memento.flush(out);

            out.close();
        }
        catch( IOException theException )
        {   JOptionPane.showMessageDialog( null, "Write Failed!",
                    "The Game of Life", JOptionPane.ERROR_MESSAGE);
        }
    }

    /** Override paint to ask the outermost Neighborhood
     *  (and any subcells) to draw themselves recursively.
     *  All knowledge of screen size is also encapsulated.
```

```
     *  (The size is passed into the outermost <code>Cell</code>.)
     */

    public void paint(Graphics g)
    {
        Rectangle panelBounds = getBounds();
        Rectangle clipBounds  = g.getClipBounds();

        // The panel bounds is relative to the upper-left
        // corner of the screen. Pretend that it's at (0,0)
        panelBounds.x = 0;
        panelBounds.y = 0;
        outermostCell.redraw(g, panelBounds, true);
    }

    /** Force a screen refresh by queuing a request on
     *  the Swing event queue. This is an example of the
     *  Active Object pattern (not covered by the Gang of Four).
     *  This method is called on every clock tick. Note that
     *  the redraw() method on a given <code>Cell</code>
     *  does nothing if the <code>Cell</code> doesn't
     *  have to be refreshed.
     */

    private void refreshNow()
    {   SwingUtilities.invokeLater
        (   new Runnable()
            {   public void run()
                {   Graphics g = getGraphics();
                    if( g == null )     // Universe not displayable
                        return;
                    try
                    {
                        Rectangle panelBounds = getBounds();
                        panelBounds.x = 0;
                        panelBounds.y = 0;
                        outermostCell.redraw(g, panelBounds, false);
                    }
                    finally
                    {   g.dispose();
                    }
                }
            }
        );
    }
}
```

The *Cell* Interface

Listing 3-8 shows Cell.java.

Listing 3-8. *Cell.java*

```
package com.holub.life;
import java.awt.*;

import com.holub.life.Storable;

/**...*/

public interface Cell
{
    /** Figure out the next state of the cell, given the specified
     *  neighbors.
     *  @return true if the cell is unstable (changed state).
     */
    boolean figureNextState(    Cell north,     Cell south,
                                Cell east,      Cell west,
                                Cell northeast, Cell northwest,
                                Cell southeast, Cell southwest );

    /** Access a specific contained cell located at the edge of the
     *  composite cell.
     *  @param row      The requested row. Must be on the edge of
     *                  the block.
     *  @param column   The requested column. Must be on the edge
     *                  of the block.
     *  @return true    if the state changed.
     */
    Cell edge( int row, int column );

    /** Transition to the state computed by the most recent call to
     *  {@link #figureNextState}
     *  @return true if a changed of state happened during the transition.
     */
    boolean transition();

    /** Redraw yourself in the indicated
     *  rectangle on the indicated Graphics object if necessary. This
     *  method is meant for a conditional redraw, where some of the
     *  cells might not be refreshed (if they haven't changed state,
     *  for example).
     *  @param g redraw using this graphics,
     *  @param here a rectangle that describes the bounds of the
     *  current cell.
```

```
     *  @parem drawAll if true, draw an entire compound cell;
     *  otherwise, draw only the subcells that need to be redrawn.
     */

    void redraw(Graphics g, Rectangle here, boolean drawAll);

    /** A user has clicked somewhere within you.
     *  @param here The position of the click relative to the bounding
     *              rectangle of the current Cell.
     */

    void userClicked(Point here, Rectangle surface);

    /** Return true if this cell or any subcells are alive.
     */
    boolean isAlive();

    /** Return the specified width plus the current cell's width
     */
    int widthInCells();

    /** Return a fresh (newly created) object identical to yourself
     *  in content.
     */
    Cell create();

    /** Returns a Direction indicated the directions of the cells
     *  that have changed state.
     *  @return A Direction object that indicates the edge or edges
     *          on which a change has occurred.
     */

    Direction isDisruptiveTo();

    /** Set the cell and all subcells into a "dead" state.
     */

    void clear();

    /**
     *  The Memento interface stores the state
     *  of a cell and all its subcells for future restoration.
     */

    interface Memento extends Storable
    {   /** On creation of the memento, indicate that a cell is
         *  alive.
```

```
        */
        void markAsAlive    (Point location);

        /** On restoration of a cell from a memento, indicate that
         *  a cell is alive.
         */
        boolean isAlive (Point location);
    }

    /**  This method is used internally to save or restore the state
     *   of a cell from a memento.
     *   @return true if this cell was modified by the transfer.
     */
    boolean transfer( Storable memento, Point upperLeftCorner,
                                                boolean doLoad );

    /** Possible value for the "load" argument to transfer() */
    public static boolean STORE = false;

    /** Possible value for the "load" argument to transfer() */
    public static boolean LOAD = true;

    /** This method is used by container of the outermost cell.
     *  It is not used internally. It need be implemented only by
     *  whatever class defines the outermost cell in the universe.
     *  Other cell implementations should throw an
     *  UnsupportedOperationException when this method is called.
     */
    Storable createMemento();

    /** The DUMMY Singleton represents a permanently dead (thus stable)
     *  cell. It's used for the edges of the grid. It's a Singleton.
     *  The Dummy class is private, but it is accessed through
     *  the public DUMMY field, declared below. I'd like this
     *  class to be private, but the JLS doesn't allow private
     *  members in an interface.
     */

    public static final Cell DUMMY = new Cell()
    {
        public boolean figureNextState(Cell n,  Cell s,  Cell e,  Cell w,
                                       Cell ne, Cell nw, Cell se, Cell sw)
                                            {return true;           }

        public Cell      edge(int r, int c) {return this;           }
        public boolean   isAlive()          {return false;          }
        public Cell      create()           {return this;           }
```

```
        public Direction isDisruptiveTo()   {return Direction.NONE;    }
        public void      clear()            {                          }
        public int       widthInCells()     {return 0;                 }
        public boolean   transition()       {return false;             }

        public void userClicked(Point h, Rectangle s                 )
        public void redraw      (Graphics g, Rectangle here,
                                                boolean drawAll    )

        public boolean transfer( Storable m, Point ul, boolean load )
        {   return false;
        }

        public Storable createMemento()
        {   throw new UnsupportedOperationException(
                            "Cannot create memento of dummy block");
        }
    };
}
```

The *Resident* Class

Listing 3-9 Shows Resident.java.

Listing 3-9. *Resident.java*

```
package com.holub.life;

import java.awt.*;
import javax.swing.*;
import com.holub.ui.Colors; // Contains constants specifying various
                            // colors not defined in java.awt.Color.
import com.holub.life.Cell;
import com.holub.life.Storable;
import com.holub.life.Direction;
import com.holub.life.Neighborhood;
import com.holub.life.Universe;

/**...*/

public final class Resident implements Cell
{
    private static final Color BORDER_COLOR = Colors.DARK_YELLOW;
    private static final Color LIVE_COLOR   = Color.RED;
    private static final Color DEAD_COLOR   = Colors.LIGHT_YELLOW;

    private boolean amAlive     = false;
    private boolean willBeAlive = false;
```

```

    private boolean isStable(){return amAlive == willBeAlive; }

    /** figure the next state.
     *  @return true if the cell is not stable (will change state on the
     *  next transition().
     */
    public boolean figureNextState(
                            Cell north,     Cell south,
                            Cell east,      Cell west,
                            Cell northeast, Cell northwest,
                            Cell southeast, Cell southwest )
    {
        verify( north,      "north"     );
        verify( south,      "south"     );
        verify( east,       "east"      );
        verify( west,       "west"      );
        verify( northeast,  "northeast" );
        verify( northwest,  "northwest" );
        verify( southeast,  "southeast" );
        verify( southwest,  "southwest" );

        int neighbors = 0;

        if( north.    isAlive()) ++neighbors;
        if( south.    isAlive()) ++neighbors;
        if( east.     isAlive()) ++neighbors;
        if( west.     isAlive()) ++neighbors;
        if( northeast.isAlive()) ++neighbors;
        if( northwest.isAlive()) ++neighbors;
        if( southeast.isAlive()) ++neighbors;
        if( southwest.isAlive()) ++neighbors;

        willBeAlive = (neighbors==3 || (amAlive && neighbors==2));
        return !isStable();
    }

    private void verify( Cell c, String direction )
    {   assert (c instanceof Resident) || (c == Cell.DUMMY)
                : "incorrect type for " + direction +  ": " +
                   c.getClass().getName();
    }

    /** This cell is monetary, so it's at every edge of itself. It's
     *  an internal error for any position except for (0,0) to be
     *  requsted since the width is 1.
     */
```

```
    public Cell edge(int row, int column)
    {   assert row==0 && column==0;
        return this;
    }

    public boolean transition()
    {   boolean changed = isStable();
        amAlive = willBeAlive;
        return changed;
    }

    public void redraw(Graphics g, Rectangle here, boolean drawAll)
    {   g = g.create();
        g.setColor(amAlive ? LIVE_COLOR : DEAD_COLOR );
        g.fillRect(here.x+1, here.y+1, here.width-1, here.height-1);

        // Doesn't draw a line on the far right and bottom of the
        // grid, but that's life, so to speak. It's not worth the
        // code for the special case.

        g.setColor( BORDER_COLOR );
        g.drawLine( here.x, here.y, here.x, here.y + here.height );
        g.drawLine( here.x, here.y, here.x + here.width, here.y  );
        g.dispose();
    }

    public void userClicked(Point here, Rectangle surface)
    {   amAlive = !amAlive;
    }

    public void    clear()         {amAlive = willBeAlive = false; }
    public boolean isAlive()       {return amAlive;                }
    public Cell    create()        {return new Resident();         }
    public int     widthInCells()  {return 1;}

    public Direction isDisruptiveTo()
    {   return isStable() ? Direction.NONE : Direction.ALL ;
    }

    public boolean transfer(Storable blob,Point upperLeft,boolean doLoad)
    {
        Memento memento = (Memento)blob;
        if( doLoad )
        {   if( amAlive = willBeAlive = memento.isAlive(upperLeft) )
                return true;
        }
        else if( amAlive )                      // store only live cells
```

```
            memento.markAsAlive( upperLeft );

        return false;
    }

    /** Mementos must be created by Neighborhood objects. Throw an
     *  exception if anybody tries to do it here.
     */
    public Storable createMemento()
    {   throw new UnsupportedOperationException(
                    "May not create memento of a unitary cell");
    }
}
```

The *Neighborhood* Class

Listing 3-10 shows Neighborhood.java.

Listing 3-10. *Neighborhood.java*

```
package com.holub.life;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import javax.swing.*;

import com.holub.io.Files;
import com.holub.life.Cell;
import com.holub.ui.MenuSite;
import com.holub.ui.Colors;
import com.holub.asynch.ConditionVariable;

import com.holub.life.Cell;
import com.holub.life.Clock;
import com.holub.life.Direction;
import com.holub.life.Storable;



/**...*/

public final class Neighborhood implements Cell
{
    /** Block if reading is not permitted because the grid is
     *  transitioning to the next state. Only one lock is
     *  used (for the outermost neighborhood) since all updates
     *  must be requested through the outermost neighborhood.
```

```
     */
    private static final ConditionVariable readingPermitted =
                                            new ConditionVariable(true);

    /** Returns true only if none of the cells in the Neighborhood
     *  changed state during the last transition.
     */

    private boolean amActive = false;

    /** The actual grid of Cells contained within this neighborhood. */
    private final Cell[][] grid;

    /** The neighborhood is square, so gridSize is both the horizontal
     *  and vertical size.
     */
    private final int      gridSize;

    /** Create a new Neigborhood containing gridSize-by-gridSize
     *  clones of the prototype. The Prototype is deliberately
     *  not put into the grid.
     */

    public Neighborhood(int gridSize, Cell prototype)
    {
        this.gridSize = gridSize;
        this.grid = new Cell[gridSize][gridSize];

        for( int row = 0; row < gridSize; ++row )
            for( int column = 0; column < gridSize; ++column )
                grid[row][column] = prototype.create();
    }

    /** The "clone" method used to create copies of the current
     *  neighborhood. This method is called from the containing
     *  neighborhood's constructor. (The current neighborhood
     *  is passed into the containing-neighborhood constructor
     *  as the "prototype" argument.
     */

    public Cell create()
    {   return new Neighborhood(gridSize, grid[0][0]);
    }

    /** Became stable on the last clock tick. One more refresh is
     *  required.
     */

    private boolean oneLastRefreshRequired = false;
```

```

    /** Shows the direction of the cells along the edge of the block
     *  that will change  state in the next transition. For example,
     *  if the upper-left corner has changed, then the current
     *  Cell is disruptive in the NORTH, WEST, and NORTHWEST directions.
     *  If this is the case, the neighboring
     *  cells may need to be updated, even if they were previously
     *  stable.
     */
    public  Direction isDisruptiveTo(){ return activeEdges; }
    private Direction activeEdges = new Direction( Direction.NONE );

    /** Figures the next state of the current neighborhood and the
     *  contained neighborhoods (or cells). Does not transition to the
     *  next state, however. Note that the neighboring cells are passed
     *  in as arguments rather than being stored internally&mdash;an
     *  example of the Flyweight pattern.
     *
     *  @see #transition
     *  @param north        The neighbor to our north
     *  @param south        The neighbor to our south
     *  @param east         The neighbor to our east
     *  @param west         The neighbor to our west
     *  @param northeast    The neighbor to our northeast
     *  @param northwest    The neighbor to our northwest
     *  @param southeast    The neighbor to our southeast
     *  @param southwest    The neighbor to our southwest
     *
     *  @return true if this neighborhood (i.e. any of it's cells)
     *              will change state in the next transition.
     */

    public boolean figureNextState( Cell north,     Cell south,
                                    Cell east,      Cell west,
                                    Cell northeast, Cell northwest,
                                    Cell southeast, Cell southwest )
    {
        boolean nothingHappened = true;

        // Is some ajacent neighborhood active on the edge
        // that ajoins me?

        if(     amActive
            ||  north    .isDisruptiveTo().the( Direction.SOUTH     )
            ||  south    .isDisruptiveTo().the( Direction.NORTH     )
            ||  east     .isDisruptiveTo().the( Direction.WEST      )
            ||  west     .isDisruptiveTo().the( Direction.EAST      )
            ||  northeast.isDisruptiveTo().the( Direction.SOUTHWEST )
            ||  northwest.isDisruptiveTo().the( Direction.SOUTHEAST )
```

```
            || southeast.isDisruptiveTo().the( Direction.NORTHWEST )
            || southwest.isDisruptiveTo().the( Direction.NORTHEAST )
        )
        {
            Cell    northCell,      southCell,
                    eastCell,       westCell,
                    northeastCell, northwestCell,
                    southeastCell, southwestCell;

            activeEdges.clear();

            for( int row = 0; row < gridSize; ++row )
            {   for( int column = 0; column < gridSize; ++column )
                {
                    // Get the current cell's eight neighbors

                    if(row == 0 )
                    {   northwestCell = (column==0)
                                ? northwest.edge(gridSize-1,gridSize-1)
                                : north.edge    (gridSize-1,column-1)
                                ;

                        northCell=  north.edge(gridSize-1,column);

                        northeastCell = (column == gridSize-1 )
                                ? northeast.edge (gridSize-1, 0)
                                : north.edge     (gridSize-1, column+1)
                                ;
                    }
                    else
                    {   northwestCell  = (column == 0)
                                ? west.edge(row-1, gridSize-1)
                                : grid[row-1][column-1]
                                ;

                        northCell = grid[row-1][column];

                        northeastCell = (column == gridSize-1)
                                ? east.edge(row-1, 0)
                                : grid[row-1][column+1]
                                ;
                    }

                    westCell = (column == 0)
                                ? west.edge( row, gridSize-1)
                                : grid[row][column-1]
                                ;

                    eastCell = (column == gridSize-1)
```

```
                        ? east.edge(row, 0)
                        : grid[row][column+1]
                        ;

                if(row == gridSize-1)
                {   southwestCell = ( column==0 )
                        ? southwest.edge(0,gridSize-1)
                        : south.edge(0,column-1)
                        ;

                    southCell = south.edge(0,column);

                    southeastCell = (column == gridSize-1 )
                        ? southeast.edge(0,0)
                        : south.edge(0, column+1)
                        ;
                }
                else
                {   southwestCell  = (column == 0)
                        ? west.edge(row+1, gridSize-1)
                        : grid[row+1][column-1]
                        ;

                    southCell = grid[row+1][column];

                    southeastCell = (column == gridSize-1)
                        ? east.edge(row+1, 0)
                        : grid[row+1][column+1]
                        ;
                }

                // Tell the cell to change its state. If
                // the cell changed (the figureNextState request
                // returned false), then mark the current block as
                // unstable. Also, if the unstable cell is on the
                // edge of the block modify activeEdges to
                //  indicate which edge or edges changed.

                if( grid[row][column].figureNextState
                    ( northCell,     southCell,
                      eastCell,      westCell,
                      northeastCell, northwestCell,
                      southeastCell, southwestCell
                    )
                )
                {   nothingHappened = false;
                }
            }
        }
```

```
        }

        if( amActive && nothingHappened )
            oneLastRefreshRequired = true;

        amActive = !nothingHappened;
        return amActive;
    }


    /** Transition the neighborhood to the previously-computed
     *  state.
     *  @return true if the transition actually changed anything.
     *  @see #figureNextState
     */
    public boolean transition()
    {
        // The condition variable is set and reset only by the
        // outermost neighborhood. It's actually incorrect
        // for an inner block to touch it because the whole
        // board has to be locked for edge cells in a subblock
        // to compute their next state correctly. There's no
        // race condition since the only place that transition()
        // is called is from the clock tick, and recursively
        // from here. As long as the recompute time is less
        // than the tick interval, everything's copasetic.

        boolean someSubcellChangedState = false;

        if( ++nestingLevel == 0 )
            readingPermitted.set(false);

        for( int row = 0; row < gridSize; ++row )
            for( int column = 0; column < gridSize; ++column )
                if( grid[row][column].transition() )
                {   rememberThatCellAtEdgeChangedState(row, column);
                    someSubcellChangedState = true;
                }

        if( nestingLevel-- == 0 )
            readingPermitted.set(true);

        return someSubcellChangedState;
    }
    // The following variable is used only by the transition()
    // method. Since Java doesn't support static local variables,
    // I am forced to declare it in class scope, but I deliberately
    // don't put it up at the top of the class definition because
    // it's not really an attribute of the class&mdash;it's just
```

```
    // an implemenation detail of the immediately preceding
    // method.
    //
    private static int nestingLevel = -1;


    /** Modifies activeEdges to indicate whether the addition
     *  of the cell at (row,column) makes an edge active.
     */
    private void rememberThatCellAtEdgeChangedState(int row,int column)
    {
        if( row == 0 )
        {   activeEdges.add( Direction.NORTH );

            if(column==0)
                activeEdges.add( Direction.NORTHWEST );
            else if(column==gridSize-1)
                activeEdges.add( Direction.NORTHEAST );
        }
        else if( row == gridSize-1 )
        {   activeEdges.add( Direction.SOUTH );

            if(column==0)
                activeEdges.add( Direction.SOUTHWEST );
            else if(column==gridSize-1)
                activeEdges.add( Direction.SOUTHEAST );
        }

        if( column == 0 )
        {   activeEdges.add( Direction.WEST );
        }
        else if( column == gridSize-1 )
        {   activeEdges.add( Direction.EAST );
        }
        // else it's an internal cell. Do nothing.
    }

    /** Redraw the current neighborhood only if necessary (something
     *  changed in the last transition).
     *
     *  @param g Draw onto this graphics.
     *  @param here Bounding rectangle for current Neighborhood.
     *  @param drawAll force a redraw, even if nothing has changed.
     *  @see #transition
     */

    public void redraw(Graphics g, Rectangle here, boolean drawAll)
    {
        // If the current neighborhood is stable (nothing changed
```

```
        // in the last transition stage), then there's nothing
        // to do. Just return. Otherwise, update the current block
        // and all sub-blocks. Since this algorithm is applied
        // recursively to subblocks, only those blocks that actually
        // need to update will actually do so.


        if( !amActive && !oneLastRefreshRequired && !drawAll )
            return;
        try
        {
            oneLastRefreshRequired = false;
            int compoundWidth = here.width;
            Rectangle subcell = new Rectangle(  here.x, here.y,
                                                here.width  / gridSize,
                                                here.height / gridSize );

            // Check to see if we can paint. If not, just return. If
            // so, actually wait for permission (in case there's
            // a race condition, then paint.

            if( !readingPermitted.isTrue() )
                return;

            readingPermitted.waitForTrue();

            for( int row = 0; row < gridSize; ++row )
            {   for( int column = 0; column < gridSize; ++column )
                {   grid[row][column].redraw( g, subcell, drawAll );
                    subcell.translate( subcell.width, 0);
                }
                subcell.translate(-compoundWidth, subcell.height);
            }

            g = g.create();
            g.setColor( Colors.LIGHT_ORANGE );
            g.drawRect( here.x, here.y, here.width, here.height );

            if( amActive )
            {   g.setColor( Color.BLUE );
                g.drawRect( here.x+1,     here.y+1,
                            here.width-2, here.height-2 );
            }

            g.dispose();
        }
        catch( InterruptedException e )
        {   // thrown from waitForTrue. Just
            // ignore it, since not printing is a
```

```
            // reasonable reaction to an interrupt.
        }
    }

    /** Return the edge cell in the indicated row and column.
     */
    public Cell edge(int row, int column)
    {   assert    (row    == 0 || row    == gridSize-1)
               || (column == 0 || column == gridSize-1)
               :  "central cell requested from edge()";

        return grid[row][column];
    }

    /** Notification of a mouse click. The point is relative to the
     *  upper-left corner of the surface.
     */
    public void userClicked(Point here, Rectangle surface)
    {
        int pixelsPerCell = surface.width / gridSize ;
        int row           = here.y       / pixelsPerCell ;
        int column        = here.x       / pixelsPerCell ;
        int rowOffset     = here.y       % pixelsPerCell ;
        int columnOffset  = here.x       % pixelsPerCell ;

        Point position = new Point( columnOffset, rowOffset );
        Rectangle subcell = new Rectangle(  0, 0, pixelsPerCell,
                                                  pixelsPerCell );

        grid[row][column].userClicked(position, subcell);
        amActive = true;
        rememberThatCellAtEdgeChangedState(row, column);
    }

    public boolean isAlive()
    {   return true;
    }

    public int widthInCells()
    {   return gridSize * grid[0][0].widthInCells();
    }

    public void clear()
    {   activeEdges.clear();

        for( int row = 0; row < gridSize; ++row )
            for( int column = 0; column < gridSize; ++column )
                grid[row][column].clear();

```

```
            amActive = false;
        }

        public boolean transfer(Storable memento, Point corner,
                                                            boolean load)
        {   int   subcellWidth  = grid[0][0].widthInCells();
            int   myWidth       = widthInCells();
            Point upperLeft = new Point( corner );

            for( int row = 0; row < gridSize; ++row )
            {   for( int column = 0; column < gridSize; ++column )
                {   if(grid[row][column].transfer(memento,upperLeft,load))
                        amActive = true;

                    Direction d =
                            grid[row][column].isDisruptiveTo();

                    if( !d.equals( Direction.NONE ) )
                        activeEdges.add(d);

                    upperLeft.translate( subcellWidth, 0);
                }
                upperLeft.translate(-myWidth, subcellWidth );
            }
            return amActive;
        }

        public Storable createMemento()
        {   Memento m = new NeighborhoodState();
            transfer(m, new Point(0,0), Cell.STORE);
            return m;
        }

        /**
         * The NeighborhoodState stores the state of this neighborhood
         * and all its sub-neighborhoods. For the moment, I'm storing
         * state with serialization, but a future modification might
         * rewrite load() and flush() to use XML.
         */

        private static class NeighborhoodState implements Cell.Memento
        {   Collection liveCells = new LinkedList();

            public NeighborhoodState( InputStream in ) throws IOException
                                                    { load(in); }
            public NeighborhoodState(               ){          }

            public void load( InputStream in ) throws IOException
            {   try
                {   ObjectInputStream source = new ObjectInputStream( in );
```

```
                liveCells = (Collection)( source.readObject() );
            }
            catch(ClassNotFoundException e)
            {   // This exception shouldn't be rethrown as
                // a ClassNotFoundException because the
                // outside world shouldn't know (or care) that we're
                // using serialization to load the object. Nothing
                // wrong with treating it as an I/O error, however.

                throw new IOException(
                            "Internal Error: Class not found on load");
            }
        }

        public void flush( OutputStream out ) throws IOException
        {   ObjectOutputStream sink = new ObjectOutputStream(out);
            sink.writeObject( liveCells );
        }

        public void markAsAlive(Point location)
        {   liveCells.add( new Point( location ) );
        }

        public boolean isAlive(Point location)
        {   return liveCells.contains(location);
        }

        public String toString()
        {   StringBuffer b = new StringBuffer();

            b.append("NeighborhoodState:\n");
            for( Iterator i = liveCells.iterator(); i.hasNext() ;)
                b.append( ((Point) i.next()).toString() + "\n" );
            return b.toString();
        }
    }
}
```

Mediator

The `Life` object instantiates only one Life-related class: the `Universe`. The instantiation (Listing 3-7 line 30) looks like this:

```
getContentPane().add( Universe.instance(), BorderLayout.CENTER);
```

As far as the `Life` class is concerned, the `Universe` is just a `JComponent` of some sort. The `Life` class has a single responsibility: main-frame creation. The only thing it cares about is that the `Universe` can be added to a `JFrame`. Since the `Universe` class extends `JComponent`, `Life` can just treat it as a `JComponent`. This way I can completely rework the user interface without impacting the code in the `Life` class.

The Universe class was declared in Listing 3-7. It's a Singleton with a private constructor that uses the declare-the-instance-as-static reification of the pattern. (The instance reference is declared on line 29, and the instance() method on line 160 returns this reference.) This method is called from only one place (the Life-class constructor), so it could be replaced by a simple constructor, but then the one-of-a-kind nature of the Universe object wouldn't be guaranteed.

The main purpose of the Universe is to serve as in intermediary between the Swing subsystem and the Life subsystem. As such, the Universe is an example of the **Mediator** pattern. ("Intermediary" would have been a better choice of pattern name.)

The main intent of Mediator is to coordinate the interaction between two different subsystems so that these subsystems don't have to interact directly with each other. Mediator also helps isolate subsystems—I may want swap out Swing to run Life on the Palm Pilot, for example—but the main responsibility of Mediator is to mediate a complex message flow.

A Mediator does not need to encapsulate all subsystem interaction, but the more interaction it encapsulates, the better the isolation between subsystems (at the cost of heavier coupling to the mediator subsystem, of course). If all interaction is through the mediator, then you can swap out an entire subsystem without affecting any of the other collaborators. In Life, I chose for the Universe to encapsulate all interaction with Swing except painting. The Resident and Neighborhood object paint themselves on the screen using Java's Graphics class, which is effectively a Mediator in its own right (sitting between your program and the underlying operating-system objects such as the Windows "device context"). The Universe mediator encapsulates all event management: It intercepts all UI events that come out of Swing and translates them into messages that the Life subsystem understands. For example, the Universe sets itself up to receive mouse-click messages on line 92 of Listing 3-7. It translates these into mouseClicked(...) messages, which are sent to the outermost cell. The only events Universe doesn't handle are the menuing events fielded by the clock subsystem, which, as you saw earlier, is built as a stand-alone subsystem so handles its own menuing, and so on.

The mediator is bidirectional (it passes messages from Life to Swing as well as the other way around). For example, a clock tick causes the Universe to ask Swing to refresh the screen if any of the cells changed state.

The Universe also controls a user interface of its own. It sets up and manages the single JPanel on which all the cells are drawn. So the Life classes are isolated from window maintenance and sizing as well. The Universe also sets up and manages the Grid menu that clears the game board and loads previously stored game states.

People often confuse Mediator with Facade. One way to tell the difference is that the users of a Mediator don't know anything about the other subsystems to which the Mediator talks (the "Colleagues"). The Life classes don't know or care that the Universe is talking to Swing. They get messages from the mediator but are unaware of the stimulus that causes the mediator to send the message. The MenuSite facade, on the other hand, doesn't hide that you're talking to the menuing subsystem; all it does is hide the complexity of that communication. Mediator may or may not simplify anything—that's not its main purpose; rather, Mediator effectively hides the existence of the other subsystem. Mediators are very active, hiding complex interactions such as event handling. Facades tend to be more passive, expanding a single message into the multiple messages required for some piece of work. Mediators are usually bidirectional, with messages flowing in both directions from the Colleagues. Facades tend to be one-directional: messages flow from the Clients into the Facade, but not in the other direction.

Composite Revisited

Now let's examine the classes that comprise the Life subsystem. Most of the real work happens in the Cell interface and Neighborhood and Resident classes, which reify several design patterns. Since you've already looked at Composite, let's start there.

The Cell interface (Listing 3-8) has the role of Component in the Composite pattern. Objects of the Resident class (Listing 3-9) comprise the Leaves in the pattern. They represent individual cells in the game. Objects of the Neighborhood class (Listing 3-10) comprise the Composites in the pattern. They comprise the interior nodes of the hierarchy.

The Neighborhood objects hold a two-dimensional array (8×8 in the current version) of Cells, declared as follows on line 41 of Listing 3-10:

```
private final Cell[][] grid;
```

Since the array is declared in terms of the Cell interface, it can hold both Resident and Neighborhood objects. Life's user interface makes this structure visible. Figure 3-11 shows the object hierarchy, and Figure 3-12 shows the UI for the entire Life "universe" (the entire grid of cells), seeded with a glider in the upper-right corner. A Neighborhood object (whose UI is the entire window) contains an 8×8 grid of Neighborhood objects (delimited on the UI by darker lines), each of which holds an 8×8 grid of Resident objects. I could nest even further to make a larger grid (a Neighborhood of Neighborhoods of Neighborhoods of Residents, for example).

What the Composite structure gives you is the ability to write the Neighborhood class in such a way that it doesn't care whether it contains a grid of Neighborhood objects or a grid of Resident objects. They all implement the Cell interface, so they can be treated identically using that interface. For example, when you ask a Neighborhood to draw itself, it asks the contained Cells to draw themselves, and then the Neighborhood draws a darker line around the entire grid of Cells. This process goes on recursively through any sub-Neighborhood objects, until you get down to the Resident, which draws itself as a yellow square with a border on two adjacent sides. If you were looking only at the drawing mechanism, this organization seems overly complex, but we'll look at other advantages shortly.

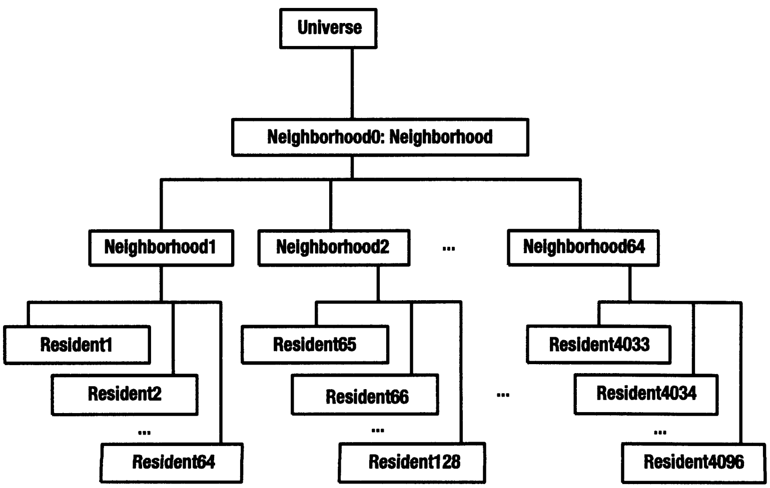


Figure 3-11. *The object hierarchy of Life*

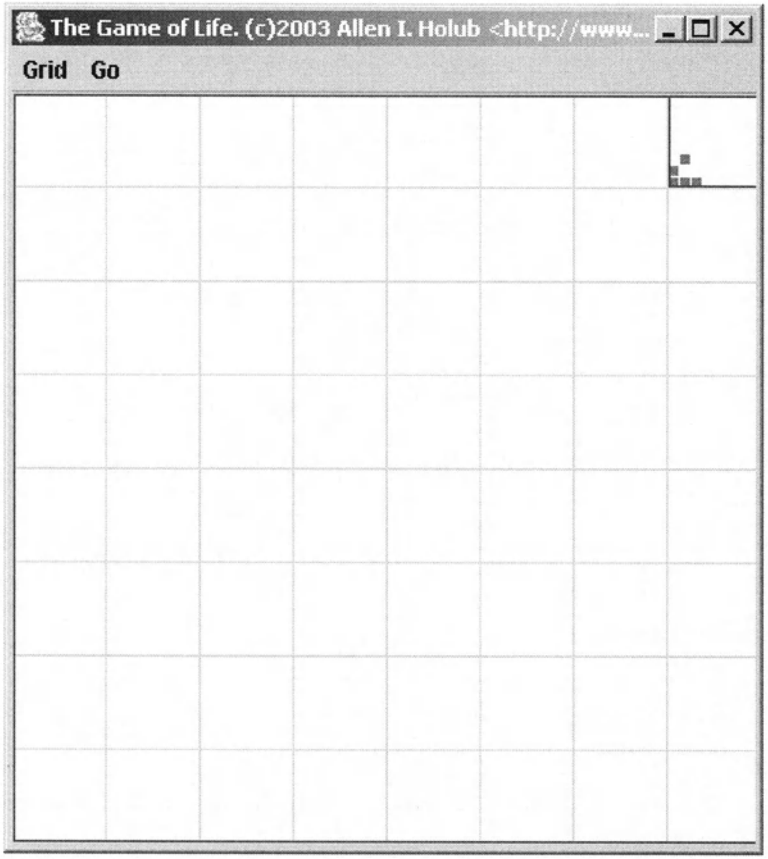


Figure 3-12. *The game board seeded with a glider*

The grid (deliberately) looks like a piece of graph paper so that you can see the object structure. The smallest squares are each drawn by a single `Cell` derivative called a `Resident`. The `Resident` has the Leaf role in the Composite pattern. (Leaves don't contain anything but their own state.) The `Neighborhood`, which holds an 8×8 array of `Cell` objects, draws itself with a darker border so that you can see its boundary.

The reason I'm using `Composite` at all is to get more efficient updates. You'll have noticed in Figure 3-12, that the `Neighborhood` that holds the glider is outlined in a darker color than the other `Neighborhoods`. Every `Cell` has a notion of "stability" associated with it. A `Cell` is stable if it will not change state on the next clock tick. A `Neighborhood` is stable if none of its contained `Cells` will change state on the next clock tick. A `Neighborhood` that is not stable displays itself with a dark-blue border. Stable `Neighborhood` objects display themselves with lighter borders. Only the unstable `Cells` are updated during clock ticks, which saves you a vast amount of work, since most `Cells` are dormant.

Figure 3-13 shows this process in action (each image is one clock tick). You can see `Neighborhood` objects become unstable as the glider moves into them. Interestingly, not every `Neighborhood` that contains a live cell is unstable; you're just interested in whether the `Cells` will change state, not whether they're in the "alive" state.

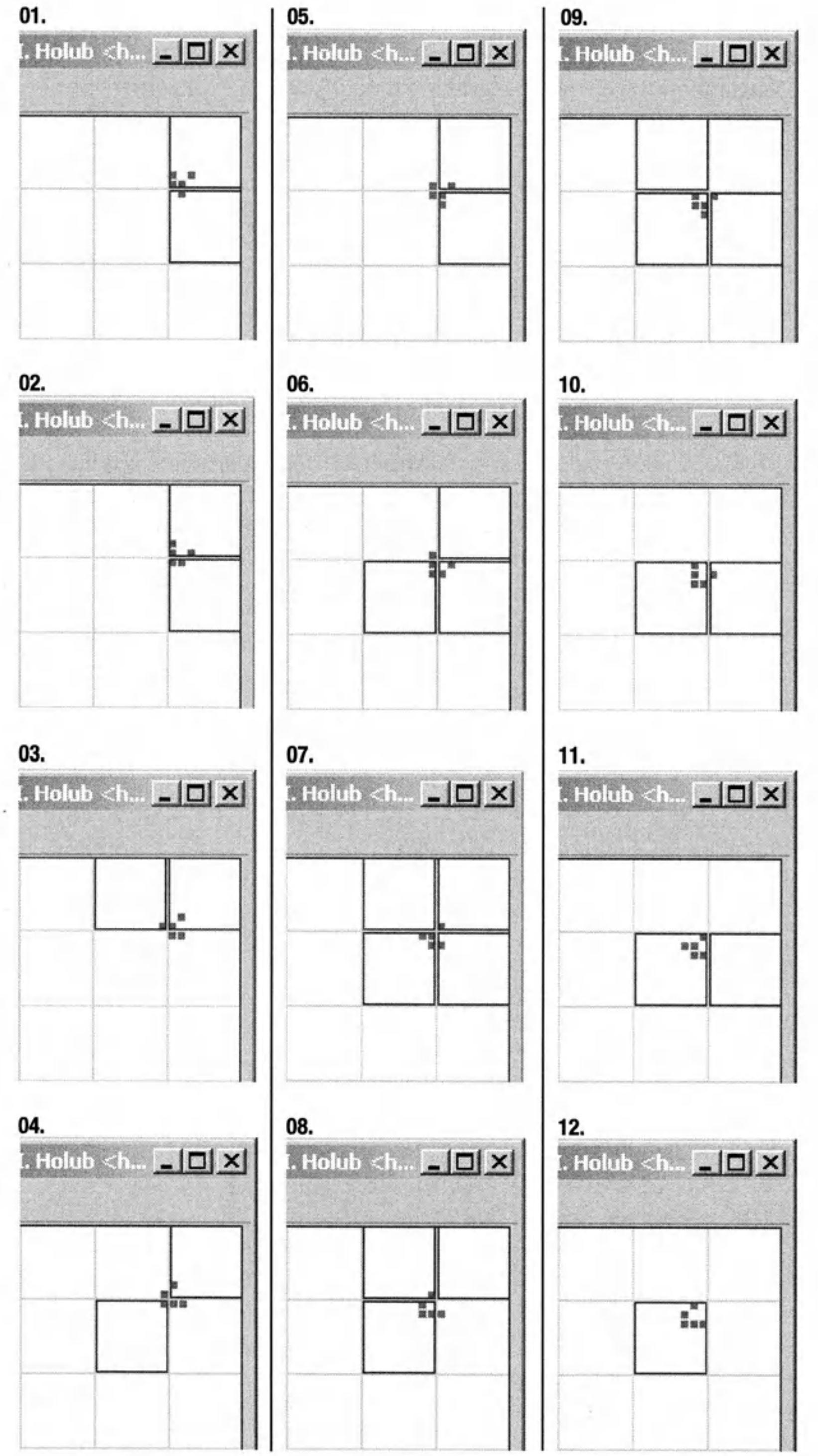


Figure 3-13. *Board behavior as glider moves*

Prototype

You can see Composite in action in Neighborhood.java by following a clock tick through the system. I'll start by looking at how the Composite grid is created. The `Universe` constructor (Listing 3-7, line 48) uses the following code to create the nested system of `Cells` that comprises the life universe:

```
outermostCell = new Neighborhood
                (   DEFAULT_GRID_SIZE,
                    new Neighborhood
                    (   DEFAULT_GRID_SIZE,
                        new Resident()
                    )
                );
```

To see what's going on here, you have to look at the `Neighborhood` constructor, but let's analyze the problem first. The `Neighborhood` doesn't know exactly what it contains (beyond that it contains `Cell` objects). Some `Neighborhood` objects will hold other `Neighborhood` objects, but others will hold `Resident` objects. The `Neighborhood` nonetheless has to manufacture the contained objects, because information that's needed to do the manufacturing (for example, the number of objects to create) is internal to the `Neighborhood`.

Two solutions spring to mind. The first is to combine the Command and Abstract Factory patterns; you pass the `Neighborhood` an Abstract Factory that knows how to create cells. The code is shown below. The Abstract Factory is also a Strategy object, since it encapsulates a creation strategy. This approach to object creation is effectively the Strategy-based approach I discussed in Chapter 2.

```
class Neighborhood
{
    interface CellFactory   // Abstract Factory Interface
    {   Cell create();
    }

    //...

    public Neighborhood( int gridSize, CellFactory factory )
    {   //...
        for( int row = 0; row < gridSize; ++row )
            for( int column = 0; column < gridSize; ++column )
                grid[row][column] = factory.create();
    }
}

//...

class Universe
{   //...
```

```
    // Pass the Neighborhood constructor an anonymous-inner-class
    // Concrete Factory that produces a <nobr><code>Cell</code></nobr> derivative.
    // (Cell is the Abstract product and either Neighborhood
    // or  Resident is the Concrete Product).

    outermostCell = new Neighborhood
                    (   DEFAULT_GRID_SIZE,
                        new Creator()
                        {   public Cell create()
                            {   return new Neighborhood
                                (   DEFAULT_GRID_SIZE,
                                    new Creator()
                                    {   public Cell create()
                                        {   return new Resident();
                                        }
                                    }
                                )
                            }
                        }
                    );
}
```

The main problem with this approach is that it's too complicated. You need an unnecessary interface (`CellFactory`), and the initialization of `outermostCell` is hideous.

The second problem is that the object you need to create may not be in a default, newly constructed state. For example, consider a runtime-customizable user interface. You can store a list of all the changes that a user has made from the default UI-object state. When you create every UI object, though, you'll have to first manufacture it and then modify its state to reflect the user preferences. You can sometimes do this modification in a constructor, but UI widgets are often provided by a third party (or by Sun as part of Java), and you don't have the option of hacking up the source code to support user customization. The create-then-modify strategy can also be quite time consuming, and the after-the-fact modifications complicates the code considerably. (A Factory is pretty much mandatory, for example.)

Here's another example: I have a generic server-side socket handler (written before the `SocketFactory` was added to Java—nowadays I'd use a `SocketFactory`). My socket handler listens on the main socket, and when a client connects, it creates a `ClientConnection` Command object to handle the actual communication with the client. Using a Command object means I don't have to use implementation inheritance to change the way the socket handler works. I just pass it an instance of some class that implements the `ClientConnection` interface. The problem is that the socket handler has to manufacture a `ClientConnection` object every time a client connects. (It actually makes a pool of `ClientConnection` objects and reuses them, but that's just an implementation detail.) I could solve this problem by passing in a `ClientConnectionFactory` object, but that approach has the same problems as the earlier example.

To the rescue comes the **Prototype** pattern: when all you have is a reference to an interface, and you need to make many instances of the referenced object, then clone them.

To solve the UI problem using Prototype, you'd serialize a user-customized version of a UI component to the disk. The next time you ran the program, you'd reload the serialized version and then make copies of that prototype object rather than calling new.

To solve the socket-connection problem, you'd pass the socket-handler constructor a prototype ClientConnection object. The socket handler will just clone the prototype on an as-needed basis.

The Neighborhood constructor uses Prototype to create subcells, using the following code:

```
public Neighborhood(int gridSize, Cell prototype)
{
    this.gridSize = gridSize;
    this.grid = new Cell[gridSize][gridSize];

    for( int row = 0; row < gridSize; ++row )
        for( int column = 0; column < gridSize; ++column )
            grid[row][column] = prototype.create();
}
```

Prototype lets you remove all knowledge of the concrete Cell-derivative type from the Neighborhood: it's passed a prototype Cell, which in practice is either a Resident or another Neighborhood, and it populates itself with clones of the prototype.

I opted to use a create() method rather than a clone() override to get type safety; clone() returns Object, so you have to cast its return value. A call to clone() works just fine if you don't mind the cast.

Composite Redux

Moving back to Composite, having populated the Neighborhood, you now need to pass messages to the cells. I'll use the clock-tick activities as an example. Figure 3-14 shows the clock-tick-initiated message flow. (The remainder of this section explains the diagram.)

The Universe Mediator translates clock ticks into the messages that cause the board to update. It subscribes to the clock-tick message as follows (Listing 3-7, line 140):

```
Clock.instance().addClockListener
(   new Clock.Listener()
    {   public void tick()
        {   if( outermostCell.figureNextState
                    ( Cell.DUMMY,Cell.DUMMY,Cell.DUMMY,Cell.DUMMY,
                      Cell.DUMMY,Cell.DUMMY,Cell.DUMMY,Cell.DUMMY
                    )
              )
            {   if( outermostCell.transition() )
                    refreshNow();
            }
        }
    }
);
```

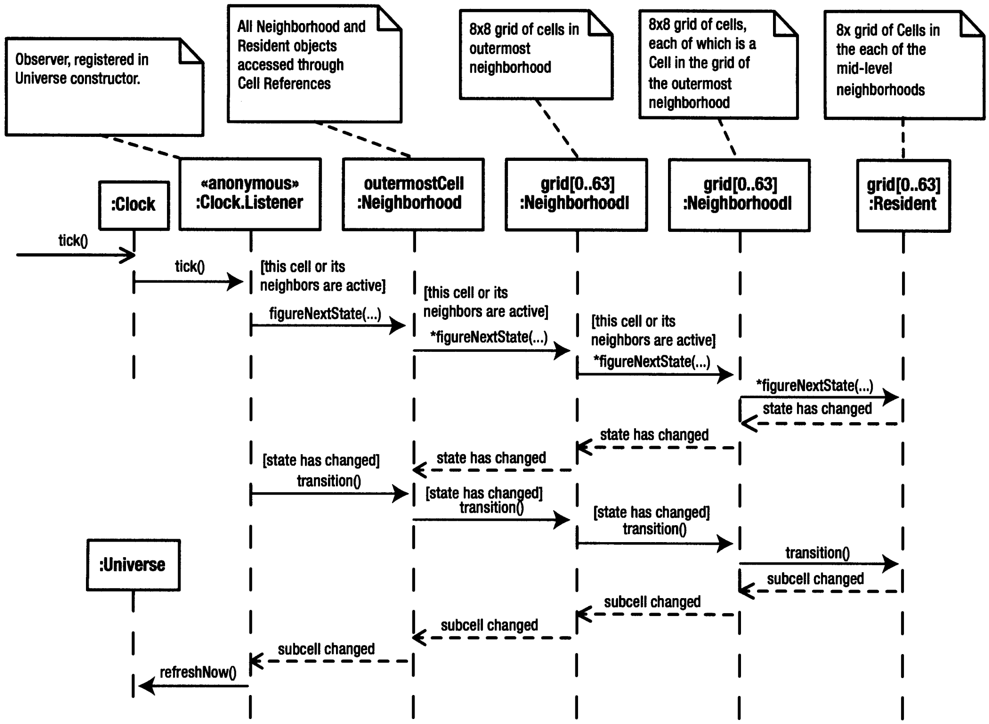


Figure 3-14. *The messages that follow a clock tick*

The message handler (`tick()`) passes a `figureNextState()` message to the outermost cell. If any of the contained cells think they may need to change state in the next pass, `figureNextState()` returns true, and the tick handler sends `transition()` message to the outermost cell to force a transition to the next state. Finally, `refreshNow()` is called to force a screen refresh if any of the contained cells actually changed state.

Starting with the `Resident`, the `figureNextState()` method (Listing 3-9, line 30) is passed references to its neighbors (more on these references later); it counts the number of live neighbors, and it determines its next state based on the neighbor count. In the second pass, the `transition()` method (Listing 3-9, line 75) just moves to that state. The `transition()` method also remembers whether it changed state for reasons that will become clear in a moment.

At the Composite level, I'll explain `transition()` first because it's simpler. Bear in mind as you read the following that the main Composite-related issue is that the `Neighborhood` doesn't have to know whether it contains other `Neighborhood` objects or whether it contains `Resident` objects. The high-level behavior (Listing 3-10, line 241) is identical. The `Neighborhood` just relays the message to its contained cells.

```
for( int row = 0; row < gridSize; ++row )
    for( int column = 0; column < gridSize; ++column )
        if( grid[row][column].transition() )
        {   rememberThatCellAtEdgeChangedState(row, column);
            someSubcellChangedState = true;
        }
```

If the subcell changed state, then the `Neighborhood` object remembers this fact and reports it to the caller.

The `Neighborhood` also keeps track of whether any cells at the edge of this `Neighborhood` have changed state, but unlike the `Resident`, the `Neighborhood` needs to keep track of which edges of the neighborhood are active. (A `Resident` doesn't bother because, if it changes state, all the edges are active.) Adjacent `Neighborhoods` need the active-edge information because the states of subcells at the adjacent-`Neighborhood`'s edges may need to change state if cells in this `Neighborhood` are actively changing. Figure 3-15 illustrates the issues. A change in cell C, for example, affects three adjacent neighborhoods (shown in grey): when cell C changes state, the cell in the southwest corner of the northeast neighborhood may need to change state as well.

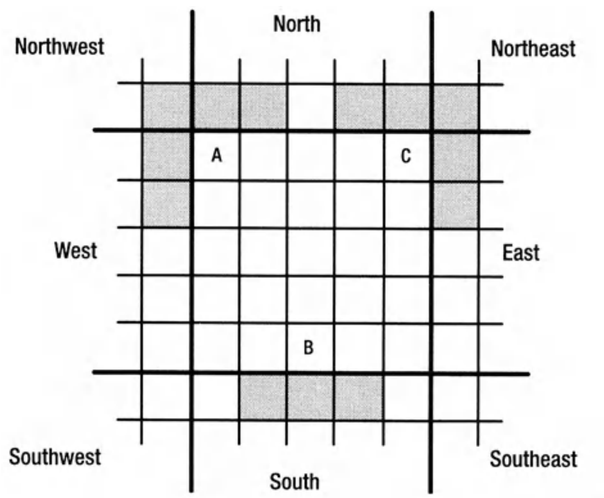


Figure 3-15. *Edge activity affects other neighborhoods*

The `rememberThatCellAtEdgeChangedState()` method (Listing 3-10, line 284) keeps track of things in a `Direction` object called `activeEdges`.

The `Direction` class (Listing 3-11) defines a simple wrapper around a bit map. The `add()` method sets a bit. The `has()` method tests that a bit is set. The oddly named `the()` method works just like `has()`. It's there so that the following call reads like English.

```
northNeighbor.isDisruptiveTo().the( Direction.SOUTH );
```

The `isDisruptiveTo` method (Listing 3-10, line 88) returns the `activeEdges Direction` you saw earlier—the one that's modified during the transition process to identify the edges of the `Neighborhood` that contains cells that have changed state in the current transition.

Listing 3-11. *Direction.java*

```
package com.holub.life;

/**...*/

```

```
public class Direction
{   private int map = BITS_NONE;

    private static final int BITS_NORTH     = 0x0001;
    private static final int BITS_SOUTH     = 0x0002;
    private static final int BITS_EAST      = 0x0004;
    private static final int BITS_WEST      = 0x0008;
    private static final int BITS_NORTHEAST = 0x0010;
    private static final int BITS_NORTHWEST = 0x0020;
    private static final int BITS_SOUTHEAST = 0x0040;
    private static final int BITS_SOUTHWEST = 0x0080;
    private static final int BITS_ALL       = 0x00ff;
    private static final int BITS_NONE      = 0x0000;

    // Various directions. Note that since we're talking
    // about the edges of a grid, NORTH | WEST and NORTHWEST are
    // different things. NORTH means that anything along the NORTH
    // edge is active; ditto for WEST and the west edge. NORTHWEST
    // means that the cell in the NORTHWEST corner is active.
    // If the NORTHWEST corner is active, the NORTH and WEST
    // edges will also be active, but the converse is not true.

    public static final Direction NORTH     = new Immutable(BITS_NORTH);
    public static final Direction SOUTH     = new Immutable(BITS_SOUTH);
    public static final Direction EAST      = new Immutable(BITS_EAST);
    public static final Direction WEST      = new Immutable(BITS_WEST);
    public static final Direction NORTHEAST = new Immutable(BITS_NORTHEAST);
    public static final Direction NORTHWEST = new Immutable(BITS_NORTHWEST);
    public static final Direction SOUTHEAST = new Immutable(BITS_SOUTHEAST);
    public static final Direction SOUTHWEST = new Immutable(BITS_SOUTHWEST);
    public static final Direction ALL       = new Immutable(BITS_ALL);
    public static final Direction NONE      = new Immutable(BITS_NONE);

    public  Direction()              {                }
    public  Direction( Direction d  ){  map = d.map; }
    private Direction( int bits     ){  map = bits;  }

    public boolean  equals( Direction d ){ return d.map == map; }
    public void     clear (             ){ map = BITS_NONE;     }
    public void     add   ( Direction d ){ map |= d.map;        }
    public boolean  has   ( Direction d ){ return the(d);               }
    public boolean  the   ( Direction d ){ return (map & d.map)==d.map; }

    private static final class Immutable extends Direction
    {
        private static final String message =
            "May not modify Direction constant (Direction.NORTH, etc.)";

```

```
53          private Immutable(int bits){ super(bits); }
54
55          public void clear()
56          {   throw new UnsupportedOperationException(message);
57          }
58
59          public void add( Direction d )
60          {   throw new UnsupportedOperationException(message);
61          }
62      }
63  }
```

The Direction implementation has a couple of other issues. Note that the bit values declared at the top of the class definition are not exposed to the outside world. The add() method, for example, takes a Direction argument, not an int that holds a bit mask. If I allowed an int argument, it would be possible for a careless programmer to pass a nonsense value into add(). Passing a Direction makes it impossible to pass add() a bad value.

The other interesting facet of the Direction class is the Immutable variant (Listing 3-11, line 48). Immutable extends Direction, overriding all methods that can modify a Direction object to throw exceptions. The prebuilt Direction objects (NORTH, SOUTH, and so on) are all instances of Immutable because a user of these objects shouldn't be modifying them. By using Immutable, I guarantee that the object can't be modified rather than leaving it up to the goodwill of the programmer. (Design note: It's been argued that I got things backward here—that a subclass shouldn't refuse to do something that the base-class contract says that it can do. It's a reasonable point, but I don't see how inverting things changes the situation.)

The Immutable class is also an example of a situation where a design-pattern solution would add more complexity than it's worth. You could implement immutability with the Decorator pattern, described in Chapter Four, but the subclass is an inner class of the class that it's extending, and it is a trivial extension to boot, so problems such as fragile base classes are immaterial.

Also note that Direction is not a Singleton because there will be many instances of it, and you can create a Direction using new. On the other hand, the eight predefined directions are very Singleton-like in their behavior. In his book *Pattern Hatching* (Addison-Wesley, 1998), John Vlissides—one of the Gang of Four—pointed out that a Singleton doesn't actually have to be limited to a single instance, as long as the number of instances is constrained. It is reasonable for a Singleton reification to manage a constrained set of instances rather than a single instance, in the same way that Direction manages a set of eight predefined Direction objects. Nonetheless, it's difficult to tell whether Direction is a Singleton simply by looking only at its structure. The public constructor is the only clue to its non-Singleton-ness.

Flyweight

The obvious way to implement Life would be to make each Cell a JButton derivative. That way, when you were setting up a pattern on the grid, you could bring a cell to life simply by clicking on it, the normal button-press mechanism can be leveraged to handle the change of appearance and state. You could arrange the buttons that represented the cells using a large JFrame and a `GridLayout` object. In this naive implementation, each button would also hold references to all eight neighbors. Though this approach is by far the easiest to implement, it's impractical. Swing components are "lightweight" only in the sense that there's no underlying OS window backing them. Looking at the JSDK 1.4.1 sources

- The `JButton` class holds two nonstatic fields.
- The `AbstractButton` superclass holds 28 nonstatic fields.
- The `JComponent` superclass holds 23 nonstatic fields.
- The `Container` superclass holds 23 nonstatic fields.
- The `Component` superclass holds 48 nonstatic fields.

That's 124 fields total—496 bytes. About half of these fields are references to other objects that are also good sized and hold references to even more objects. Let's guess conservatively and assume that each of these referenced objects requires 50 bytes, yielding another 3,100 bytes. You also need to add 8 pointers to the Cell's neighbors and a boolean to remember the current cell state (36 bytes). So, the grand total is 3,632 bytes per button. To make the math easy, let's assume that the life "Universe" is a 1024×1024 grid of cells. That's an even 1,048,576 cells. Multiplying by the cell size, you get 3.6 gigabytes (3,632MB) of memory required to hold the grid. Odds are, you don't have that much core memory in your machine, which means that the array will have to be stored in virtual memory and paged into core as the program runs. This paging is an extremely time-consuming process. The net result would be excruciatingly slow performance.

Obviously, the obvious approach won't work.

I've solved the problem by combining Composite with another design pattern, **Flyweight**.

The notion of a flyweight is tied closely to the definition of an object. If you've read somewhere that an object is a bundle of data and a set of "method" functions that access the data, then you've been misled. This sort of description is typical of a procedural programmer who's new to objects, but it's fundamentally incorrect. An object is defined primarily by what it does, by the messages that it can handle. The object will typically have some sort of internal state represented by a set of fields, but the way in which this state is implemented internally has absolutely nothing to do with what the object is. All that should matter are the methods.

Don't be confused here by the notion of "attributes." At the risk of repeating something I said in Chapter 1, an *attribute* is a characteristic of an object that serves to distinguish a class of objects from another class of objects. A "salary" attribute, for example, distinguishes one class of people (employee) from other classes of people (volunteer, consultant, former dot-com-er, and so on). The most important attributes of the object are the methods—the set of messages that the object can handle. Other attributes serve as a design aid that tells you whether a method makes sense. (Asking a volunteer to `printYourSalary()` won't do anything useful since a volunteer doesn't have a salary attribute.)

Simply because an object has an attribute does not mean that it has an associated field. *Synthesized* attributes are computed at runtime, not stored in the object, for example. A salary attribute, may be inferred from a pay grade, a title, years of employment, or some other measure that was stored as a field. It may be computed from a complex formula that involved fields, method calls, and database lookups.

The attribute-related issue that concerns the Flyweight pattern is that all the attributes of some class of object may be synthesized. The fact that the class has no fields in it does not impact its "object-ness" in any way, as long as it has responsibilities (and the methods needed to exercise those responsibilities). Moreover, it's often debatable where a particular attribute should be stored. Take the `Neighborhood` and `Resident` as a case in point. It's reasonable for a `Resident` to know its size and position on the screen. By the same token, it's equally reasonable for a `Neighborhood` to know the size and positions of all the `Cells` it contains. Generally, you'd put this information into the `Resident` class because it would be easier for a `Resident` object to draw itself. It's not "wrong," however, for a `Neighborhood` object to synthesize a `Cell`'s size and position and pass that information to a contained `Cell`. If done properly (by accessing subcells through an interface), a design that moves the size and location of an element into the container doesn't tighten the coupling at all.

When a class of objects allows a container class to hold data that could just as easily be stored in the contained object, then the data is called *extrinsic data*. For example, the `figureNextState()` method of the `Cell` interface you looked at earlier is passed references to the `Cell`'s eight neighbors. The neighbor references could be contained inside the `Cell`, but that would take too much space at runtime. By the same token, it's perfectly reasonable that a container such as a `Neighborhood` would be able to synthesize the eight neighbor references when it asks a subcell to figure its next state. Since they're external to a `Resident`, the neighbor references are considered extrinsic. The `Cell`'s size and location information are also synthesized by the surrounding `Cell`, so they are also extrinsic. In fact, only two fields of the `Resident` class are not extrinsic: the `amAlive` and `willBeAlive` fields (declared on lines 21 and 22 of Listing 3-9). The current implementation stores this information in `boolean` fields, but I could save even more space by setting and clearing bits in a `byte` instead of using two `booleans`.

The `Cell`, then, is a Flyweight. (*Flyweight* is a term for a boxer who weighs less than 112 pounds.) Most of a Flyweight's state information is extrinsic. You can see how the extrinsic data in a flyweight works by following a mouse click from the `Universe` (which manages the only window in the system) to the `Resident` that has to service the click. Figure 3-16, explained shortly, shows how the messages propagate when the mouse is clicked from the position shown in Figure 3-17 (ten cells from both the top and left edge of the universe).

A `Resident` isn't a window because of the memory requirements, and its size and location are extrinsic for the same reason. The mouse-click handler in the `Universe` class (Listing 3-7, line 92) sends the outermost cell a `userClicked()` message, passing as arguments the window-relative position of the mouse "hot spot" and a bounding rectangle—a `Rectangle` whose horizontal and vertical size is the size of the window and whose upper-left corner is at position (0,0). The outermost cell is actually being passed the size of itself (the outermost cell is as big as the whole window) and the location within itself of the mouse click.

Since the outermost cell is a `Neighborhood`, this call gets you to the `userClicked()` override in the `Neighborhood` class (Listing 3-10, line 390). This override relays the message to its subcells. First it figures out which subcell contains the click position. `pixelsPerCell` holds the number of pixels in a subcell (the container cell width divided by the number of cells.) Using this information and the click location, the override determines which subcell needs to be informed of the click and relays the message to that subcell only.

The important thing to notice is that the calling method passes the subcell a rectangle that identifies the subcell's size (the number of pixels in a single subcell), and the calling method modifies the click position to be relative to the subcell's bounding rectangle.

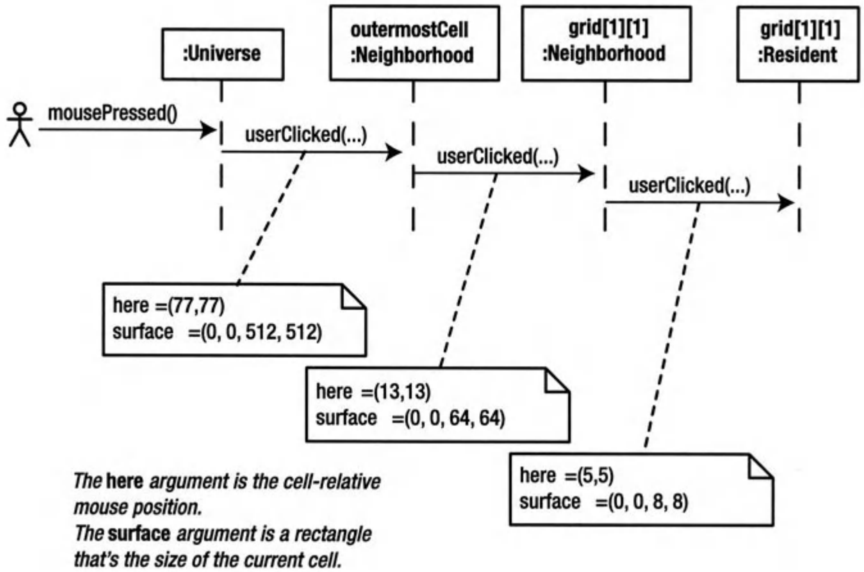


Figure 3-16. *The messages that follow a mouse click*

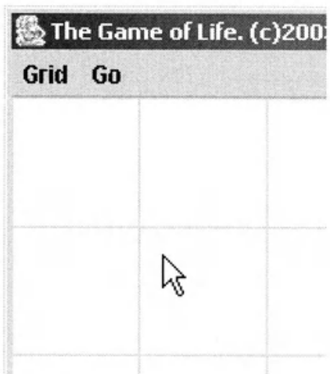


Figure 3-17. *Mouse position for scenario in Figure 3-16*

Since the outermost `Neighborhood` contains other `Neighborhood` objects, this first call to `userClicked(...)` (on line 402) is actually recursive. It's received by the contained `Neighborhood` object. This contained object, then, does the same work, scaling the size of the rectangle down even further and moving the position to be relative to its subcell's bounding rectangle. The method calls `userClicked(...)` again, but this time the contained `Cell` is a `Resident`, so you end up in the override in Listing 3-9, line 96. The `Resident` doesn't care where, within itself, the click occurred, so the `Resident` version of `userClicked(...)` ignores its arguments. The method just inverts its `amAlive` state.

Screen painting happens in a similar way. The `Universe` sends `redraw(...)` messages to the outermost cell on lines 221 and 245 of Listing 3-7, getting you to the `Neighborhood` override in Listing 3-10, line 321. This override scales down the rectangle and relays it to the subcell (on line 352), which eventually gets you to the `Resident` override (Listing 3-9, line 81), which draws one cell in yellow or red, depending on whether it's alive.

You should note two other things in this code. A `Neighborhood` draws a darker-than-usual border around itself when it's not stable. This is the code that generated the moving outlines you looked at earlier as the glider flew across the universe. This last example is another example of why it's a good thing for an object to display its own UI. It would be a lot harder to do this "outlining" in an external rendering class.

Also, the `Neighborhood`'s `redraw(...)` override (Listing 3-10, line 321) doesn't do anything if the test at the top succeeds. That is, if the current `Neighborhood` is stable, then the version on the screen is just fine, so the `Neighborhood` doesn't redraw itself. This same logic applies to the `figureNextState()` (Listing 3-10, line 111). If the `Neighborhood` is stable, it doesn't ask the contained cells to figure their states. This way, you don't waste machine cycles updating cells that don't need to be updated.

To finish with this aspect of Flyweight, you can see the dark underbelly of the pattern in the `Neighborhood` class's implementation of `figureNextState()` (Listing 3-10, line 111). This method is made hideously complicated by the fact that the neighbors of cells on the edge of a `Neighborhood` are in a different `Neighborhood` object. All that nasty code after line 144 of Listing 3-10 is just figuring out which neighborhood holds the adjacent cell. None of this complexity would be necessary if the cells held their own neighbor pointers, but getting rid of this excess baggage was the whole point of using Flyweight to begin with.

The `figureNextState()` method makes many calls to `edge(...)`, which returns a cell on the edge of an adjacent `Neighborhood`. The `edge()` method looks an awful lot like one of the getters I disparaged earlier in this chapter, so some explanation is in order. Remember, the basic argument against getters is that they expose implementation details and negatively impact maintainability. Here, however, the cells are a fundamental attribute of a `Neighborhood`. The fact that a `Neighborhood` is made up of `Cells` is one of the key defining characteristics (attributes) of a neighborhood. As I mentioned in Chapter 1, occasionally providing method-level access to a core attribute is at times okay, and this is one of those times. It would be a serious error to expose how the `Neighborhood` stores the cells, but it's harmless to expose the fact that the `Neighborhood` simply contains cells.

Moreover, `edge(...)` is called only by other `Neighborhood` objects. Passing data between two identical objects doesn't impact maintenance one iota, since they both instantiate the same class definition. Normally, I'd make a method such as `edge(...)` `private` to ensure that it wasn't called from foreign classes, but I can't do that here because the Composite pattern mandates access through the `Cell` interface. I could get around this problem by dispensing with the `Cell` interface, making `edge(...)` private, and redefining `Resident` to extend `Neighborhood` and override all the public methods. Although this reorganization lets me restrict access to `edge(...)`, it's unacceptable for a `Cell` to carry around all the baggage of a `Neighborhood` (the array of subcells, for example) when it's not using that baggage.

Flyweight Pools

Returning to the clock-tick handler, as follows, the `Universe` passes the outermost cell eight references to the `Cell.DUMMY` object:

```
Clock.instance().addClockListener
(   new Clock.Listener()
    {   public void tick()
        {   if( outermostCell.figureNextState
                    ( Cell.DUMMY,Cell.DUMMY,Cell.DUMMY,Cell.DUMMY,
                      Cell.DUMMY,Cell.DUMMY,Cell.DUMMY,Cell.DUMMY
                    )
              )
            {   if( outermostCell.transition() )
                    refreshNow();
            }
        }
    }
);
```

This code is a simplistic example of *flyweight pooling,* the other main characteristic of the Flyweight pattern. Rather than create eight identical flyweights, I use eight references to a single flyweight. (In fact, the `DUMMY` object actually masquerades as 256 instances of `Cell`, since the eight references passed into `figureNextState()` are themselves treated as `Neighborhood` objects, each of which uses the same `DUMMY` objects as the cells on the edge of its `Neighborhood`.) Conceptually, the entire grid that comprises the Life universe is bordered by `DUMMY` objects, but these objects on the border are all actually the same object.

The `DUMMY` object is defined using the anonymous-inner-class mechanism in the `Cell` interface (Listing 3-8, line 128). It implements a dead cell that does nothing. By passing it into the outermost `Cell` of the composite, this cell is effectively surrounded by "dummy" objects. Using an anonymous inner class makes the actual class definition inaccessible.

The `DUMMY` object, by the way, is yet another Singleton. The instance is manufactured in the `DUMMY` declaration on line 128 of Listing 3-8. The Singleton is accessed globally using `Cell.DUMMY` instead of an accessor method. Only one instance of the class can possibly exist since the class itself is defined using the anonymous inner-class mechanism. You can't create another instance using `new` because you don't have a class name to use.

A better example of Flyweight pooling is Java's `BorderFactory` class. The `javax.Swing.Border` defines a Flyweight, albeit a big one. The `Border` interface defines a paint method that uses four variables to render the border. Here's the prototype for that method:

```
void paintBorder(Component c, Graphics g, int x, int y, int width, int height)
```

Since all these arguments could just as well be attributes of the class that implements `Border`, these arguments really define the `Border`'s extrinsic data. Making these fields extrinsic yields an important benefit. A single `Border` object can draw borders around any numbers of components. For example, I can put a three-pixel `EmptyBorder` around several components with the following code:

```
Border threePixelPadding = new EmptyBorder( 3, 3, 3, 3 );

JButton hello   = new JButton("Hello");
JButton goodbye = new JButton("Goodbye");

hello.setBorder( threePixelPadding );
goodbye.setBorder( threePixelPadding );
```

The extrinsic information needed to render the border is passed into the Border threePixelPadding object when it's time to do the drawing.

Since the border is so flexible, and since most Border objects are used around many components, there's no real need to use new to make a Border derivative with certain characteristics every time you need one. That is, it's better to use the same Border object everywhere rather than to create many identical objects. You want to cache a single instance and use the instance in the cache.

Swing accomplishes caching with an Abstract Factory: BorderFactory. Use it like this:

```
JButton hello   = new JButton("Hello");
JButton goodbye = new JButton("Goodbye");

hello.setBorder  ( BorderFactory.createEmptyBorder(3,3,3,3) );
goodbye.setBorder( BorderFactory.createEmptyBorder(3,3,3,3) );
```

If I were implementing BorderFactory, I'd do it as a flyweight pool. The first time I was asked for a three-pixel-wide empty border, I would have the BorderFactory manufacture it. Subsequent requests for Border objects with the same characteristics would return the same object. Only one three-pixel-wide-empty-Border object would exist. (Swing gives no guarantee that the BorderFactory actually works this way, so you can't safely do things such as use == to compare factory-generated objects—something that you could do if being a flyweight pool was part of the object's contract.)

So far, this code is just a reification of Abstract Factory that's used to create Singletons. (You can argue with me about whether the manufactured Border objects are indeed Singletons, but I think of them in a similar light as the Class-object Singleton.) What makes the BorderFactory a Flyweight pool is that the Singleton that's managed by the factory is a flyweight, and the purpose of the factory is to limit the number of flyweight instances to the minimal set.

Memento

One final design pattern exists in Life: **Memento**. I briefly discussed Memento in the context of OLE in-place activation back in the section "The Menuing System: Facade and Bridge." The idea of a memento is that some object (an Originator) needs another object (a Caretaker) to hold the Originator's state. The Originator encapsulates that state into a black box (a Memento), which the Caretaker stores. The Caretaker cannot modify the state of the Originator by manipulating the Memento, however. To enforce the black-box nature of a Memento, it is often represented physically as a byte array or an Object whose concrete class is unknown to the Caretaker.

You'll remember that the OLE container uses Memento to store the state of the in-place activated object. When Excel (the Originator) shuts down, it passes its state to Word as a memento—a byte array that Word stores until Excel needs again. Since Word (the Caretaker) has no idea what's in that byte array, Word can't do anything with the Memento but store it. Another good example is a web-browser cookie—a chunk of data provided by the server that the browser holds onto until it talks to that server again. The browser has no idea what's in the cookie—it's the Caretaker.

A Caretaker can store the memento as a blob in a database, by serializing it to disk or just by holding it in memory until the Originator needs it again.

Though you may think you can use Memento to implement "undo," it's not usually suitable for that purpose. Simply restoring some piece of the program to a previous state doesn't undo any "side effects" of the original operation. For example, if an object updates a database during some operation, simply restoring the object to its former state does not reverse the database update. In any event, the memento may not store actual state information; it may contain some "key" you use to get the actual state information. In JSP, for example, the cookie holds a "session ID" that's used to find the actual session state in the server. There's not enough information in the cookie itself to do anything like an undo operation. The Command pattern, discussed in Chapter Four, solves the undo problem.

In the case of Life, I wanted to be able to save the state of a Life universe (all of the cells) so that I could seed a complex pattern onto the board only once and then load the pattern back into the game at some future time. I wanted to isolate the mechanics of storage and retrieval in my `Universe` mediator, so I implemented persistence by having the `Universe` ask the `Cells` for a memento that the `Universe` stores and retrieves.

I applied two levels of interfaces in Life's implementation of Memento to guarantee the black-box quality of the Memento itself. At the `Universe` level everything is done in terms of the `Storable` interface (Listing 3-12). It has only two methods, `load()` and `flush()`, which do the obvious.

If you look back at the `Universe` (Listing 3-7), you'll see that it sets up menu handlers that store and load the entire game board (on line 113). These handlers call `doStore()` and `doLoad()` to do the actual work.

The `doStore` method (Listing 3-7, line 186) uses Abstract Factory to create a Memento.

```
Storable memento = outermostCell.memento();
```

(This Abstract Factory isn't called out in Figure 3-2 simply because there wasn't enough room to cram it in, so I've put it in Figure 3-18. `Cell` is an Abstract Factory of `Storable` Abstract Products. `Neighborhood` is the Concrete Factory of `NeighborhoodState` Concrete Products.) `doStore()` then asks the outermost cell to transfer its state into the memento. Finally, it asks the memento to flush itself out to the disk.

The `doLoad()` method (Listing 3-7, line 164) is basically the same as `doStore()`. It reverses the disk access and transfer operations, however. It first loads the memento from the disk and then asks the outermost cell to import the memento into itself.

At the Caretaker level (the `Universe`) the Memento is a black box—a `Storable` object of some sort that knows how to load and store itself. The Universe can't change the state of the data in the memento.

Listing 3-12. *Storable.java*

```
package com.holub.life;
import java.io.*;

/**...*/

public interface Storable
{   void load ( InputStream in   ) throws IOException;
    void flush( OutputStream out ) throws IOException;
}
```

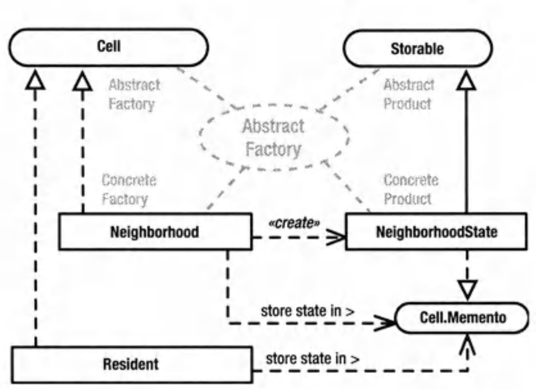


Figure 3-18. *Life's mementos*

Moving into the concrete classes, at the Life level, access to the Memento is through the `Cell.Memento` interface (Listing 3-8, line 87), implemented by the `Neighborhood.NeighborhoodState` class (Listing 3-10, line 462). `NeighborhoodState` implements the `Storable` interface to serialize itself out to the disk and back in using the built-in serialization system. At some point, I plan to replace the serialization with XML so that I can build seed files in an ASCII editor, but for the time being, serialization will do. Note that this change to an XML format involves a localized change in the `NeighborhoodState` class; it affects no other classes. `NeighborhoodState` encapsulates a linked list of points, each identifying a live cell on the board. All cells not in the list are dead. A `Resident` object marks itself as alive by calling `markAsAlive`, which simply adds a point to the list. When loading from the Memento, a `Resident` asks if it `isAlive()`, and if an affirmative answer comes back, the `Resident` object sets its state to "alive" (in the `transfer(...)` overload, Listing 3-9, line 109).

This implementation of Memento seems complex, but it has two important characteristics: I can change the way in which the game state is stored by changing only one class (`NeighborhoodState`), and, because I have isolated the Memento generation from the file system in the mediator, I can change the location of the stored Memento without changing anything except the `Universe` class. All likely changes are localized to a single class.

Loose Ends

Listings 3-13, 3-14, and 3-15 contain the remaining classes in the Game of Life.

The `Colors` interface (Listing 3-13) contains nothing but symbolic constants that alias various `java.awt.Color` values I use regularly. Use this interface like a Singleton. That is, use `Colors.DARK_RED` to access the dark-red `Color`. Don't implement the `Colors` interface to use `DARK_RED` without the prefix. Many Java programs do implement interfaces to access static data in this way, but I don't think much of that practice from a design point of view. An employee is not a color (which implies `extends`), and employees do not support messages that are passed to colors generally (which implies `implements`), so an `Employee` class should not implement `Colors`. It's better to think of `Colors` as a kind of multiway Singleton that provides global access to a constrained set of objects. Just use the fields directly.

The `Files` utility contains only one method that makes it a little easier to display a file-chooser dialog. When you want a user-selected file, you call this:

```
File in = Files.userSelected(".",".txt","Text File","Open");
```

The method takes care of the mechanics of getting the dialog box displayed. This class is a simplistic example of Facade.

Finally, the `ConditionVariable` class in Listing 3-15 is a roll-your-own threading primitive that corrects an omission in Java's `wait()` method. One of the main problems with `wait()` is that the thing you're waiting on has no notification state. That is, a thread that needs to wait for some event to occur may not want to be suspended if the event has already occurred when the thread calls `wait()`. `ConditionVariable` solves the problem by incorporating a `boolean` that's checked prior to issuing the `wait()` request. Think of a condition variable as a `boolean` that represents a condition of some sort. If the condition is false, then you wait for it to become true. If the condition is true, then you'll never wait at all. You create a condition variable in the `false` state like this:

```
ConditionVariable eventHappened = new ConditionVariable( false );
```

You can issue the following call to wait for the condition to become true:

```
eventHappened.waitForTrue();
```

When the event does happen, the event handler sets the condition variable to the true state as follows:

```
eventHappened.set( true );
```

Any waiting threads are released, and all subsequent calls to `eventHappened.waitForTrue()` return immediately without blocking. If you need the threads to start waiting for the condition variable again, set it back to a false state as follows:

```
eventHappened.set( false );
```

`ConditionVariable` is another simple Facade, simplifying a tiny bit of behavior of Java's threading subsystem.

I use a condition variable in Life to make sure that the activities associated with a clock tick don't overlap. The semaphore (`readingPermitted`) is declared at the top of the `Neighborhood` class (Listing 3-10, line 31). The reading-permitted state is set and cleared in the `Neighborhood`'s `transition()` override (Listing 3-10, line 241). Finally, the `Neighborhood`'s `redraw` override does nothing if reading is not permitted (Listing 3-10, line 345). The `waitForTrue()` on the line following this last test is just insurance that handles a potential race condition in the code.

Listing 3-13. *Colors.java*

```
// &copy; 2003 Allen I Holub. All rights reserved.
package com.holub.ui;
import java.awt.*;

/*  The Colors interface contains nothing but symbolic constants for various
 *  color values that I use regularly. The names are self explanatory.
 */
```

```

/**...*/

public interface Colors
{
/**...*/ static final Color DARK_RED      = new Color(0x99, 0x00, 0x00);
/**...*/ static final Color MEDIUM_RED    = new Color(0xcc, 0x00, 0x00);
/**...*/ static final Color LIGHT_RED     = new Color(0xff, 0x00, 0x00);

/**...*/ static final Color DARK_ORANGE   = new Color(0xff, 0x66, 0x00);
/**...*/ static final Color MEDIUM_ORANGE = new Color(0xff, 0x99, 0x00);
/**...*/ static final Color LIGHT_ORANGE  = new Color(0xff, 0xcc, 0x00);
/**...*/ static final Color ORANGE     = new Color(0xff, 0x99, 0x00);

/**...*/ static final Color OCHRE         = new Color(0xcc, 0x99, 0x00);
/**...*/ static final Color DARK_YELLOW   = new Color(0xff, 0xff, 0x00);
/**...*/ static final Color MEDIUM_YELLOW = new Color(0xff, 0xff, 0x99);
/**...*/ static final Color LIGHT_YELLOW  = new Color(0xff, 0xff, 0xdd);

/**...*/ static final Color DARK_GREEN    = new Color(0x00, 0x66, 0x00);
/**...*/ static final Color MEDIUM_GREEN  = new Color(0x00, 0x99, 0x00);
/**...*/ static final Color LIGHT_GREEN   = new Color(0x00, 0xff, 0x00);
/**...*/ static final Color GREEN         = MEDIUM_GREEN;

/**...*/ static final Color DARK_BLUE     = new Color(0x00, 0x00, 0x99);
/**...*/ static final Color MEDIUM_BLUE   = new Color(0x00, 0x00, 0xcc);
/**...*/ static final Color LIGHT_BLUE    = new Color(0x00, 0x00, 0xff);

/**...*/ static final Color DARK_PURPLE   = new Color(0x99, 0x00, 0x99);
/**...*/ static final Color MEDIUM_PURPLE = new Color(0xcc, 0x00, 0xff);
/**...*/ static final Color LIGHT_PURPLE  = new Color(0xcc, 0x99, 0xff);
/**...*/ static final Color PURPLE     = MEDIUM_PURPLE;
}
```

Listing 3-14. *Files.java*

```
package com.holub.io;

import java.io.*;
import javax.swing.*;
import javax.swing.filechooser.FileFilter; // disambiguate from java.io version

/**...*/

public class Files
{
    /** Throw up a file chooser and return the file that the user selects.
     *  @param extension File extension you're looking for. Use null if
     *                   any will do.
```

```
 *  @param description the description of what the extension means.
 *                    Not used if "extension" is null.
 *  @param selectButtonText Replaces the "Open" on the chooser button.
 *  @param startHere Name of initial directory in which to look.
 *  @return the selected file.
 *  @throws FileNotFoundException if the user didn't select a file. I've
 *          done this rather than returning null so that it's easy to
 *          do the following:
 *  <PRE>
 *  FileInputStream in =
 *      new FileInputStream(
 *                    Files.userSelected(".",".txt","Text File","Open"));
 *  </PRE>
 */

public static File userSelected( final String startHere,
                    final String extension,
                    final String description,
                    final String selectButtonText )
                                throws FileNotFoundException
{   FileFilter filter =
        new FileFilter()
        {   public boolean accept(File f)
            {   return f.isDirectory()
                        || (extension != null
                                && f.getName().endsWith(extension) );
            }
            public String getDescription()
            {   return description;
            }
        };

    JFileChooser chooser = new JFileChooser(startHere);
    chooser.setFileFilter(filter);

    int result = chooser.showDialog(null,selectButtonText);
    if(result == JFileChooser.APPROVE_OPTION)
        return chooser.getSelectedFile();

    throw new FileNotFoundException("No file selected by user");
}

static class Test
{
    public static void main(String[] args)
    {
        try
```

```
            {   File f=Files.userSelected(".",".test","Test File","Select!");
                System.out.println( "Selected " + f.getName() );
            }
            catch( FileNotFoundException e)
            {   System.out.println( "No file selected" );
            }
            System.exit(0); // Required to stop AWT thread & shut down.
        }
    }
}
```

Listing 3-15. *ConditionVariable.java*

```
package com.holub.asynch;

/**
 *  This class is a simplified version of the com.asynch.Condition
 *  class. Use it to wait for some condition to become true:
 *  <PRE>
 *  ConditionVariable hellFreezesOver = new ConditionVariable(false);
 *
 *  Thread 1:
 *      hellFreezesOver.waitForTrue();
 *
 *  Thread 2:
 *      hellFrezesOver.set(true);
 *  </PRE>
 *  Unlike <code>wait()</code> you will not be suspended at all if you
 *  wait on a true condition variable. You can call <code>set(false)</code>,
 *  to put the variable back into a false condition (thereby forcing
 *  threads to wait for it to become true, again).
 */

public class ConditionVariable
{
    private volatile boolean isTrue;

    public ConditionVariable( boolean isTrue ){ this.isTrue = isTrue; }

    public synchronized boolean isTrue()
    {   return isTrue;
    }

    public synchronized void set( boolean how )
    {   if( (isTrue = how) == true )
            notifyAll();
    }
```

```

    public final synchronized void waitForTrue() throws InterruptedException
    {   while( !isTrue )
            wait();
    }
}
```

Summing Up

Whew! That's 11 design patterns—the 9 pictured in Figure 3-2 plus Command and Abstract Factory—used in a program that has only 20 classes and interfaces in it, some of which are trivial. Though Life is a small program, it nicely demonstrates how the patterns all work together in the real world. They never stand in splendid isolation, as they would appear in a catalog-based design-patterns book.

More important, if you factor out all of the text in this chapter that describes what the pattern is, you'll find that there's hardly anything left. That is, if you knew the patterns already, I could have explained the entire Life program to you in a couple pages. This economy of expression makes for very productive conversations.

One of the main reasons for doing design at all is improved communication (between programmers, between designers and programmers, between programmers and users, and so on). I hope I've shown you how effective the design-pattern vocabulary can be in achieving that end.

This chapter also shows you one of the significant disadvantages of a hard-core design-pattern approach. As I mentioned earlier, my implementation of Life is probably the most complicated implementation of Life ever written. As I said at the beginning of the chapter, this Game of Life is, after all, a toy, and I let myself go nuts with the patterns. The current implementation certainly shows how the patterns all interact to get work done, however, and that was one of the main things I was trying to show you.