Chapter 13 Graphical User Interfaces

Chapter Objectives

After studying this chapter, you should be able to:

- Write a graphical user interface using existing Java components
- Implement interfaces using the Model-View-Controller pattern
- Structure a graphical user interface using multiple views
- Write new components for use in graphical user interfaces

A graphical user interface (GUI) often gives us the first glimpse of a new program. The information it displays indicates the program's purpose, whereas a quick review of the interface's controls and menus gives us a feel for what the program can do.

Graphical user interfaces operate in a fundamentally different way from text-based interfaces. In a text-based interface, the program is in control, demanding information when it suits the program rather than the user. With a graphical user interface, the user has much more control; users can perform operations in their preferred order rather than according to the program's demands. Naturally, this difference requires structuring the program in a different way.

This chapter pulls together the graphical user interface thread running through each chapter and adds new material, enabling us to design and build graphical user interfaces for our programs.

13.1 Overview

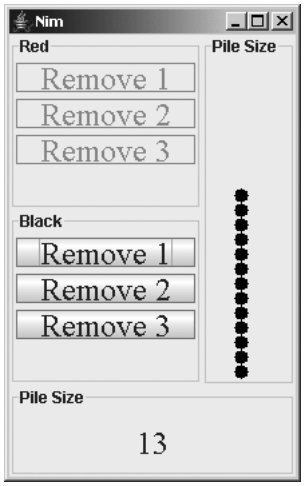
Building the graphical user interface (GUI) for a program can be one of the more rewarding parts of programming. Finally, we begin to *see* the results of our labor and are able to manipulate our program directly. The user interface is also a place where we can use aesthetic skills and sensibilities.

On the other hand, creating GUIs can involve a lot of time and frustration. Developing them will call upon every skill we've learned so far: extending existing classes, writing methods, using collaborating classes and instance variables, using Java interfaces, and so on. However, following a concrete set of steps will make the job easier. Watch for patterns that occur repeatedly. Master those patterns, and you'll be able to write GUIs like a professional.

We will proceed by developing a variant of the game of Nim. The requirements are specified in Figure 13-1.

A game of Nim begins with a pile of tokens. Two players take turns removing one, two, or three tokens from the pile. The last player to remove a token wins the game. The players will be designated "red" and "black." The first one to move will be chosen randomly. The initial size of the pile is between ten and twenty tokens and is set randomly.

An example of one possible user interface is shown on the right.

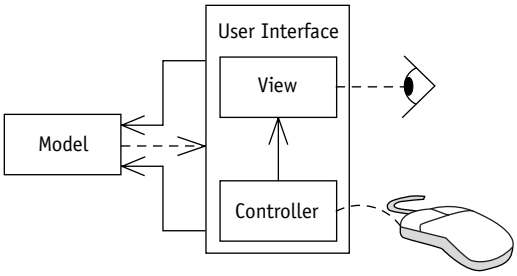


(figure 13-1)

Requirements for the game of Nim

13.1.1 Models, Views, and Controllers

Recall from Chapter 8 that graphical user interfaces are usually structured using the Model-View-Controller pattern. Figure 13-2, reproduced here from Section 8.6.2, shows the core ideas.



(figure 13-2)

View and controller interact with the user and the model

The model is the part of the program that represents the problem at hand. In our game of Nim, it's the model that will keep track of how many tokens remain on the pile, whose turn it is to move next, and who (if anyone) has won the game. The model also enforces rules. For example, it will not allow a player to take more than three tokens.

KEY IDEA

The model maintains relevant information, the view displays it, and the controller requests changes to it.

The user interface is composed of the view and the controller. The user, represented by the eye and the mouse, uses the view to obtain information from the model. It's the view, for example, that displays the current size of the pile and whose turn it is. The user interacts with the controller to change the model. In the case of Nim, the controller is used to remove some tokens or to start a new game.

The arrow between the controller and the view indicates that the controller will need to call methods in the view. The lack of an arrow going the other way indicates that the view will generally not need to call the controller's methods. The two arrows between the user interface and the model indicate that both the view and the controller will have reason to call the model's methods—the view to obtain information to display and the controller to tell the model how the user wants it to change. The dotted arrow from the model to the user interface indicates that the model will be very restrictive in how it calls methods in the interface. Essentially, it will call only a single method to tell the view that it has changed and that the view needs to update the display.

The interaction of the controller, model, and view may seem complicated at first. However, it follows a standard pattern, which includes the following typical steps, performed in the following order:

- The user manipulates the user interface—for example, enters text in a component.
- The user interface component notifies its controller by calling a method that we write.
- The controller calls a mutator method in the model, perhaps supplying additional information such as text that was entered in the component.
- Inside the mutator method, the model changes its state, as appropriate. Then it calls the view's `update` method, informing the view that it needs to update the information it displays.
- Inside the `update` method, the view calls accessor methods in the model to gather the information it needs to display. It then displays that information.

Our first graphical user interface will use a single view and controller. We will learn in Section 13.5, however, that using multiple views and controllers can actually make an interface easier to build. We will plan for that possibility from the beginning.

KEY IDEA

Interfaces usually have more than one view.

13.1.2 Using a Pattern

Models, views, and controllers make up a pattern that occurs repeatedly. The steps for using this pattern are shown in Figure 13-3. You'll find that many of the steps are familiar from previous chapters in the book. None of this is truly new material; it just puts together what we have already learned in a specific way, resulting in a graphical user interface.

Model-View-Controller

Set up the Model and View

1. Write three nearly empty classes:
   a. The model, implementing `becker.util.IModel`.
   b. The view, extending `JPanel` and implementing `becker.util.IView`. The constructor takes an instance of the model as an argument.
   c. A class containing a `main` method to run the program.
2. In `main`, create instances of the model and the view. Display the view in a frame.

Build and Test the Model	**Build the View and Controllers**
1. Design, implement, and test the model. In particular, a. add commands used by the controllers to change the model b. add queries used by the views to obtain the information to display 2. Call `updateAllViews` just before exiting any method that changes the model's state.	1. Design the interface. 2. Construct the required components and lay them out in the view. 3. Write `updateView` to update the information displayed by the view to reflect the model. 4. Write appropriate controllers for each of the components that update the model. Register the controllers.

(figure 13-3)

Steps for building a graphical user interface

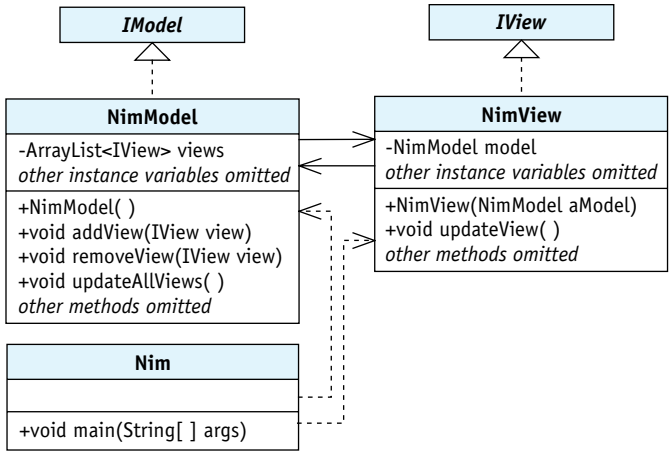
We will elaborate on these steps in each of the next three subsections.

13.2 Setting up the Model and View

The first step sets up the basic architecture for the Model-View-Controller pattern. This is where the connections between the classes are established, and by the end of this step, we will have a program that we can run, even though it won't do anything more than show us an empty frame. The class diagram of the resulting program is shown in Figure 13-4.

(figure 13-4)

Partial class diagram for `Nim`



13.2.1 The Model's Infrastructure

The model's primary purpose is to model the problem, in our case the game of Nim. It must also inform the views each time the model changes (and therefore the view needs to change the information it displays). It is this update function that we are focusing on now.

LOOKING BACK

`ArrayList`s were discussed in Section 8.5.1, interfaces in Section 7.6.

It's possible that a model may have several views, and we will provide for that possibility right away by keeping a list of views that we need to inform of changes. These requirements are embodied in the `IModel` interface. It specifies that a model needs to be able to add a view, remove a view, and update all views. The model will only need to call one method in the views, `updateView`. It expects each view to implement the `IView` interface.

KEY IDEA

The `IModel` interface specifies methods needed in the model to manage views.

A class with this infrastructure is shown in Listing 13-1. Every model will start out just like this except that the name of the class, the constructor, and the class documentation will change to reflect the program's purpose.

FIND THE CODE

ch13/nim Infrastructure/

Listing 13-1: *The model's class with infrastructure to inform views of changes*

```
1  import becker.util.IModel;
2  import becker.util.IView;
3  import java.util.ArrayList;
4
5  /** A class implementing a version of Nim. There is a (virtual) pile of tokens. Two
6  *   players take turns removing 1, 2, or 3 tokens. The player who takes the last token
7  *   wins the game.
8  *
9  *   @author Byron Weber Becker */
10 public class NimModel extends Object implements IModel
```

Listing 13-1: *The model's class with infrastructure to inform views of changes* (continued)

```
11  { private ArrayList<IView> views = new ArrayList<IView>();
12
13    /** Construct a new instance of the game of Nim. */
14    public NimModel()
15    { super();
16    }
17
18    /** Add a view to display information about this model.
19    *  @param view The view to add. */
20    public void addView(IView view)
21    { this.views.add(view);
22    }
23
24    /** Remove a view that has been displaying information about this model.
25    *  @param view The view to remove. */
26    public void removeView(IView view)
27    { this.views.remove(view);
28    }
29
30    /** Inform all the views currently displaying information about this model that the
31    *  model has changed and their display may need changing too. */
32    public void updateAllViews()
33    { for (IView view : this.views)
34      { view.updateView();
35      }
36    }
37  }
```

Of course, more must be added to `NimModel`. In particular, it does nothing yet to model the game of Nim. But when one of the players takes some tokens from the pile, for example, we now have the infrastructure in place to inform all of the views that they need to update the information they are showing the players.

Using `AbstractModel`

These three methods are always required to implement a model. Instead of writing them each time we create a model class, we can put them in their own class. Our model can simply extend that class.

Such a class, `AbstractModel`, is in the `becker.util` package. Its code is almost exactly like the code in Listing 13-1 except for the name of the class. `NimModel` is then implemented as follows:

```
import becker.util.AbstractModel;

public class NimModel extends AbstractModel
{
  public NimModel()
  { super();
  }

  // Other methods will be added here to implement the model.
}
```

`AbstractModel` implements `IModel`, implying that `NimModel` also implements that interface. The clause `implements IModel` does not need to be repeated.

The Java library has a class named `Observable` that is very similar to `AbstractModel`. It is designed to work with an interface named `Observer` that is very similar to `IView`. Why don't we use them instead? There are two reasons.

First, the `update` method in `Observable` is more complex than we need.

Second, and more importantly, the Java library doesn't have an interface corresponding to `IModel`. Therefore, the model must always extend `Observable`. Sometimes this isn't a problem (as with `NimModel`), but other times the model must extend another class. In those situations, the missing interface is required, and these classes can't be used.

At the time of this writing, Java library contains 6,558 classes. A number of those classes define their own versions of `Observer` and `Observable`, as we have done. It's interesting to note that none of the classes use `Observer` and `Observable`.

13.2.2 The View's Infrastructure

KEY IDEA

A component is nothing more than an object designed for user interfaces. Buttons, scroll bars, and text fields are all examples of components.

Each view will be a subclass of `JPanel`[1] that contains the user interface components required to interact with the model. For now, however, we will provide only the infrastructure for updating the view. That consists of implementing the `IView` interface, which specifies the `updateView` method called by the model in `updateAllViews`. This is all shown in Listing 13-2.

[1] This is true most of the time. It's convenient for menus to extend `JMenuBar` and toolbars to extend `JToolBar`.

The view is passed an instance of the model when it is constructed. The model is saved in an instance variable, and the view adds itself to the model's list of views. Finally, the view must update the information it displays by calling `updateView` in line 16.

Listing 13-2: *The view's class set up to receive notification of changes in the model*

```
1  import javax.swing.JPanel;
2  import becker.util.IView;
3
4  /** Provide a view of the game of Nim to a user.
5  *
6  *  @author Byron Weber Becker */
7  public class NimView extends JPanel implements IView
8  { private NimModel model;
9
10   /** Construct the view.
11   *  @param aModel The model we will be displaying. */
12   public NimView(NimModel aModel)
13   { super();
14     this.model = aModel;
15     this.model.addView(this);
16     this.updateView();
17   }
18
19   /** Called by the model when it changes. Update the information this view displays. */
20   public void updateView()
21   {
22   }
23 }
```

FIND THE CODE

ch13/nim Infrastructure/

13.2.3 The `main` Method

The last step in setting up the infrastructure is to write the `main` method. It constructs an instance of the model and an instance of the view. It then displays the view in an appropriately sized frame. This is shown in Listing 13-3.

FIND THE CODE

ch13/nim Infrastructure/

Listing 13-3: *The* `main` *method for running the program*

```
1  import javax.swing.JFrame;
2
3  /** Run the game of Nim. There is a (virtual) pile of tokens. Two players take turns
4  *   removing 1, 2, or 3 tokens. The player who takes the last token wins the game.
5  *
6  *   @author Byron Weber Becker */
7  public class Nim
8  {
9    public static void main(String[] args)
10   { NimModel model = new NimModel();
11     NimView view = new NimView(model);
12
13     JFrame f = new JFrame("Nim");
14     f.setSize(250, 200);
15     f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16     f.setContentPane(view);
17     f.setVisible(true);
18   }
19 }
```

13.3 Building and Testing the Model

Figure 13-3 describes the steps for building a user interface. It suggests that the model requires commands to change its state and queries for the views to use in updating the display. If we keep this in mind while using the development process discussed in Chapter 11, we will discover that the model for Nim needs the following methods:

- `removeTokens`, a command to remove one, two, or three tokens from the pile
- `getPileSize`, a query returning the current size of the pile
- `getWhoseTurn`, a query returning whose turn it is
- `getWinner`, a query returning which player, if any, has won the game

The requirements in Figure 13-1 specify that the first player and the initial size of the pile are chosen randomly. The default constructor will do that, but since randomness makes the class hard to test, we'll also add a private constructor, allowing our test harness to easily specify the pile size and first player.

LOOKING BACK

Enumerations were discussed in Section 7.3.4.

Representing the two players is a perfect job for an enumeration type. We will use three values: one for the red player, one for the black player, and one for nobody. The last one might be used, for example, as the answer to the query of who has won the game (if the game isn't over yet, nobody has won).

The `Player` enumeration is shown in Listing 13-4, and the `NimModel` class is shown in Listing 13-5. In `NimModel`, the only method (other than the constructors) that changes the model's state is `removeTokens`. After it has made its changes, it calls `updateAllViews` at line 96 to inform the views that they should update the information they display.

KEY IDEA

Call `updateAllViews` *before returning from a method that changes the model.*

FIND THE CODE

ch13/nimOneView/

Listing 13-4: *The* `Player` *enumeration type*

```
1  /** The players in the game of Nim, plus NOBODY to indicate situations where
2  *   neither player is applicable (for example, when no one has won the game yet).
3  *
4  *   @author Byron Weber Becker */
5  public enum Player
6  { RED, BLACK, NOBODY
7  }
```

FIND THE CODE

ch13/nimOneView/

Listing 13-5: *The completed* `NimModel` *class*

```
1  import becker.util.AbstractModel;
2  import becker.util.Test;
3
4  /** A class implementing a version of Nim. There is a (virtual) pile of tokens. Two
5  *   players take turns removing 1, 2, or 3 tokens. The player who takes the last token
6  *   wins the game.
7  *
8  *   @author Byron Weber Becker */
9  public class NimModel extends AbstractModel
10 { // Extending AbstractModel is an easy way to implement the IModel interface.
11
12    // Limit randomly generated pile sizes and how many tokens can be removed at once.
13    public static final int MIN_PILESIZE = 10;
14    public static final int MAX_PILESIZE = 20;
15    public static final int MAX_REMOVE = 3;
16
17    private int pileSize;
18    private Player whoseTurn;
19    private Player winner = Player.NOBODY;
20
21    /** Construct a new instance of the game of Nim. */
22    public NimModel()
23    { // Call the other constructor to do the initialization.
24      this(NimModel.random(MIN_PILESIZE, MAX_PILESIZE),
25           NimModel.chooseRandomPlayer());
26    }
```

Listing 13-5: *The completed* `NimModel` *class* (continued)

```
27
28    /** We need a way to create a nonrandom game for testing purposes. */
29    private NimModel(int pileSize, Player next)
30    { super();
31      this.pileSize = pileSize;
32      this.whoseTurn = next;
33    }
34
35    /** Generate a random number between two bounds. */
36    private static int random(int lower, int upper)
37    { return (int)(Math.random()*(upper-lower+1)) + lower;
38    }
39
40    /** Choose a player at random.
41    *  @return Player.RED or Player.BLACK with 50% probability for each */
42    private static Player chooseRandomPlayer()
43    { if (Math.random() < 0.5)
44      { return Player.RED;
45      } else
46      { return Player.BLACK;
47      }
48    }
49
50    /** Get the current size of the pile.
51    *  @return the current size of the pile */
52    public int getPileSize()
53    { return this.pileSize;
54    }
55
56    /** Get the next player to move.
57    *  @return Either Player.RED or Player.BLACK if the game has not yet been won,
58    *   or Player.NOBODY if the game has been won. */
59    public Player getWhoseTurn()
60    { return this.whoseTurn;
61    }
62
63    /** Get the winner of the game.
64    *  @return Either Player.RED or Player.BLACK if the game has already been won;
65    *  Player.NOBODY if the game is still in progress. */
66    public Player getWinner()
67    { return this.winner;
68    }
```

Listing 13-5: *The completed* `NimModel` *class* (continued)

```
69
70    /** Is the game over?
71    *  @return true if the game is over; false otherwise. */
72    private boolean gameOver()
73    { return this.pileSize == 0;
74    }
75
76    /** Remove one, two, or three tokens from the pile. Ignore any attempts to take
77    *  too many or too few tokens. Otherwise, remove howMany tokens from the pile
78    *  and update whose turn is next.
79    *  @param howMany How many tokens to remove.
80    *  @throws IllegalStateException if the game has already been won */
81    public void removeTokens(int howMany)
82    { if (this.gameOver())
83      { throw new IllegalStateException(
84                                        "The game has already been won.");
85      }
86
87      if (this.isLegalMove(howMany))
88      { this.pileSize = this.pileSize - howMany;
89        if (this.gameOver())
90        { this.winner = this.whoseTurn;
91          this.whoseTurn = Player.NOBODY;
92        } else
93        { this.whoseTurn =
94                    NimModel.otherPlayer(this.whoseTurn);
95        }
96        this.updateAllViews();
97      }
98    }
99
100   // Is howMany a legal number of tokens to take?
101   private boolean isLegalMove(int howMany)
102   { return howMany >= 1 && howMany <= MAX_REMOVE &&
103            howMany <= this.pileSize;
104   }
105
106   // Return the other player.
107   private static Player otherPlayer(Player who)
108   { if (who == Player.RED)
109     { return Player.BLACK;
110     } else if (who == Player.BLACK)
111     { return Player.RED;
```

Listing 13-5: *The completed* `NimModel` *class* (continued)

```
112       } else
113       { throw new IllegalArgumentException();
114       }
115     }
116
117     // The addView, removeView, and updateAllViews methods could be included
118     // here. That isn't necessary in this case because NimModel extends AbstractModel.
119
120     /** Test the class. */
121     public static void main(String[] args)
122     { System.out.println("Testing NimModel");
123       NimModel nim = new NimModel(10, Player.RED);
124       Test.ckEquals("pile size", 10, nim.getPileSize());
125       Test.ckEquals("winner", Player.NOBODY, nim.getWinner());
126       Test.ckEquals("next", Player.RED, nim.getWhoseTurn());
127
128       /** ------ find the code to see complete test suite ------*/
129     }
130 }
```

13.4 Building the View and Controllers

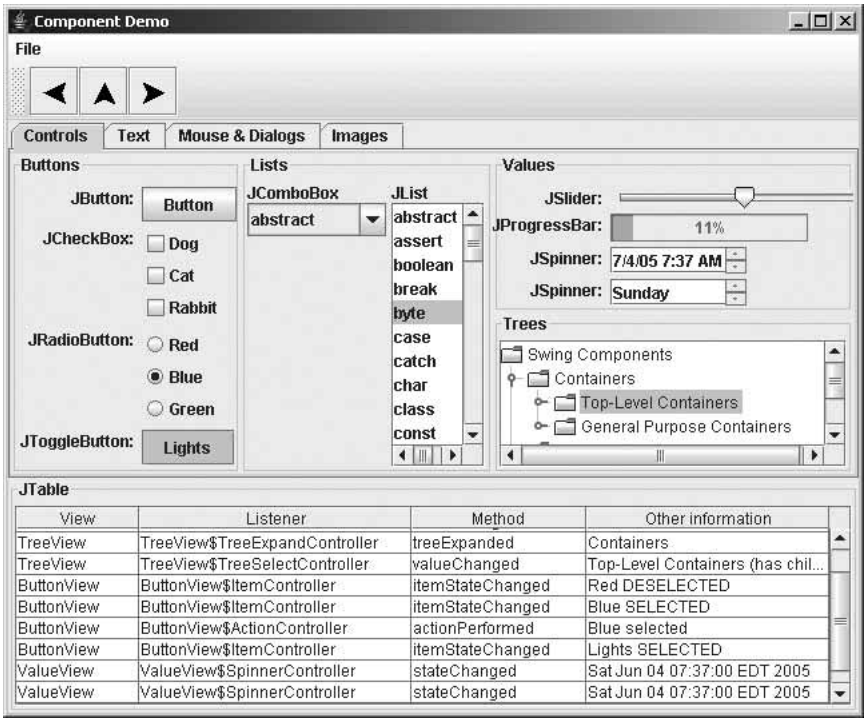
The view, of course, is what displays information from the model to the user. It is the visible part of the user interface. The controllers are what make the interface interactive. They listen for the user manipulating controls such as buttons or menus and then make appropriate calls to the commands in the model.

13.4.1 Designing the Interface

KEY IDEA

The program shown in Figure 13-5 contains lots of code to help get you started using components.

Java comes with many user interface components including buttons, text fields, menus, sliders, and labels. Some of these are shown in Figure 13-5. Designing an interface includes deciding which of these components are most appropriate both to display the model and to accept input from the user, and how to best arrange them on the screen. For now, while we're learning the basics, we will restrict ourselves to labels for displaying information and text fields to accept input from the user. In Section 13.7, we will explore other components.



(figure 13-5)

Application demonstrating many of the components available for constructing views

FIND THE CODE

ch13/componentDemo/

Our first view will appear as shown in Figure 13-6. It shows the end of the game after Red has won. The text areas (one has "2" in it, the other has "3") are enabled when it's the appropriate player's turn and disabled when it isn't. When the game is over, both are disabled, as shown here.

(figure 13-6)

First view for the game of Nim

13.4.2 Laying Out the Components

The components for any view can be divided into those that require ongoing access and those that don't. In this view, the following five components require ongoing access either to change the information they display or to obtain changes made by the user.

- Two `JTextFields` to accept input from the players
- One `JLabel` showing the pile's current size
- Two `JLabels` announcing the winner (they are not visible until there is a winner, and even then only one is shown)

KEY IDEA

References to components requiring ongoing access are stored in instance variables.

References to these objects will be stored in instance variables.

```
// Get how many tokens to remove.
private JTextField redRemoves = new JTextField(5);
private JTextField blackRemoves = new JTextField(5);

// Info to display.
private JLabel pileSize = new JLabel();
private JLabel redWins = new JLabel("Winner!");
private JLabel blackWins = new JLabel("Winner!");
```

LOOKING BACK

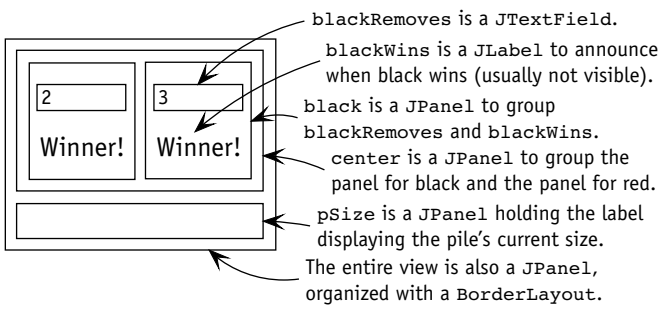
Layout managers were discussed in Section 12.6.

The components that do not require ongoing access include several `JPanel` objects used to organize the components and the borders around them. Instance variables storing references to these components are not required.

These components are laid out using four nested `JPanels`, as shown in Figure 13-7.

(figure 13-7)

`NimView` *uses nested* `JPanels` *to lay out the components*



The task of laying out the components occurs when the view is constructed and is usually complex enough to merit a helper method called from the constructor. We'll call the helper method `layoutView`, as shown in Listing 13-6. The method carries out the following tasks:

- The first `JPanel`, named `red`, is defined in lines 12–15. It contains a `JTextField` to accept information from the red player and a label to announce if red is the winner. The `JPanel` itself is wrapped with a border to label it in line 15.
- The second `JPanel`, `black`, is just like `red` except that it contains components for the black player.
- The third `JPanel`, `pSize`, contains the label used to display the size of the pile. It, too, has a border to label it.
- The fourth `JPanel`, `center`, is not directly visible in the user interface. It exists solely to group the `red` and `black` `JPanel`s into a single component that can be placed as a whole.

Finally, recall that `NimView` is itself a `JPanel` that can have its own layout manager. It is set in line 36 to be a `BorderLayout`. Only two of the layout's five areas are used, the center and the south side. The center section grows and shrinks as its container is resized. That's where we put the `center` panel containing `red` and `black`. The south area contains `pSize`.

Adding the `layoutView` method to `NimView`, as shown in Listing 13-6, and running the program results in something that looks much like Figure 13-6. The pile size won't be displayed and both players will be declared winners. To display that information correctly we need to update the view with information from the model.

Listing 13-6: *A helper method to lay out the view for Nim*

```
1  public class NimView extends JPanel implements IView
2  { // Instance variables omitted.
3
4    public NimView(NimModel aModel)
5    { // Details omitted.
6      this.layoutView();
7    }
8
9    // Layout the view.
10   private void layoutView()
11   { // A panel for the red player.
12     JPanel red = new JPanel();
13     red.add(this.redRemoves);
14     red.add(this.redWins);
```

FIND THE CODE

ch13/nimOneView/

Listing 13-6: *A helper method to lay out the view for Nim* (continued)

```
15        red.setBorder(BorderFactory.createTitledBorder("Red"));
16
17        // A panel for the black player.
18        JPanel black = new JPanel();
19        black.add(this.blackRemoves);
20        black.add(this.blackWins);
21        black.setBorder(BorderFactory.createTitledBorder("Black"));
22
23        // Pile size information.
24        JPanel pSize = new JPanel();
25        pSize.add(this.pileSize);
26        pSize.setBorder(
27              BorderFactory.createTitledBorder("Pile Size"));
28
29        // Group the red and black panels.
30        JPanel center = new JPanel();
31        center.setLayout(new GridLayout(1, 2));
32        center.add(red);
33        center.add(black);
34
35        // Lay out the pieces in this view.
36        this.setLayout(new BorderLayout());
37        this.add(center, BorderLayout.CENTER);
38        this.add(pSize, BorderLayout.SOUTH);
39    }
40 }
```

13.4.3 Updating the View

KEY IDEA

The `updateView` *method is responsible for updating the view with the latest information from the model.*

The `updateView` method was already added when we set up the model and view architecture, but it doesn't do anything yet. It is called by the model each time the model changes so that it can update the view's components with current information.

For the moment, we want `updateView` to perform three basic tasks:

- Display the correct pile size.
- Enable the `JTextField` for the red player when it is the red player's turn and disable it otherwise, with similar behavior for the black player's text field. When a component is disabled, the players can't use it, thus forcing each player to take his or her turn at the right time.

- Make `redWins` visible when the red player wins the game and invisible when it hasn't, with similar behavior for `blackWins`.

Recall that the constructor received a reference to the model as a parameter. This reference was stored in an instance variable named, appropriately, `model`. We will use it to retrieve the necessary information from the model to carry out these tasks.

Updating the Size of the Pile

The component to display the size of the pile is a `JLabel`. It has a method, `setText`, which takes a string and causes the label to display it. Thus, we can update the pile size display with the following statement:

```
this.pileSize.setText("" + this.model.getPileSize());
```

The result from `getPileSize` is an `int`. "Adding" it to the empty string forces Java to convert it to a string, which is what `setText` requires.

If you run the program now, the user interface should show the pile size.

Updating the Text Fields

`redRemoves` is the name of the text field used by the red player to say how many tokens to remove. To enable or disable it, we'll use the `setEnabled` method, passing `true` to enable the component and `false` to disable it. We want the text field enabled when the following Boolean expression is `true`:

```
this.model.getWhoseTurn() == Player.RED
```

If this expression is `false` (it's not red's turn), the component should be disabled. Thus,

```
this.redRemoves.setEnabled(
                this.model.getWhoseTurn() == Player.RED);
```

enables `redRemoves` when it's the red player's turn and disables it otherwise. Recall that when the game is over, `getWhoseTurn` returns `Player.NOBODY`, resulting in both text fields being disabled.

Updating the Winners

When the game is over, we want either `redWins` or `blackWins` to become visible. If the game isn't over, we want both to be invisible. Every component has a method named `setVisible` that makes the component visible when passed the value `true`

and invisible when passed the value `false`. We can again use a simple Boolean expression to pass the correct value:

```
this.redWins.setVisible(
            this.model.getWinner() == Player.RED);
```

LOOKING AHEAD

We will refine `updateView` *in Section 13.4.5.*

A similar statement for `blackWins` completes the method. Like `getWhoseTurn`, `getWinner` can also return `Player.NOBODY`.

The entire method is shown in Listing 13-7. If you run the program with this method completed, the user interface should display the initial pile size, one of the text fields should be enabled (indicating who removes the first tokens), and neither player should have their "Winner!" label showing. However, the game still can't be played because the components will not yet respond to the users.

FIND THE CODE

ch13/nimOneView/

Listing 13-7: *Updating the view with current information from the model*

```
1  public class NimView extends JPanel implements IView
2  { private NimModel model;
3    private JTextField redRemoves = new JTextField(5);
4    // Other instance variables, constructor, and methods omitted.
5
6    /** Called by the model when it changes. Update the information this view displays. */
7    public void updateView()
8    { // Update the size of the pile.
9      this.pileSize.setText("" + this.model.getPileSize());
10
11     // Enable and disable the text fields for each player.
12     this.redRemoves.setEnabled(
13             this.model.getWhoseTurn() == Player.RED);
14     this.blackRemoves.setEnabled(
15             this.model.getWhoseTurn() == Player.BLACK);
16
17     // Proclaim the winner, if there is one.
18     this.redWins.setVisible(
19             this.model.getWinner() == Player.RED);
20     this.blackWins.setVisible(
21             this.model.getWinner() == Player.BLACK);
22   }
23 }
```

13.4.4 Writing and Registering Controllers

The fundamental job of a controller is to detect when a user is manipulating a component and to respond in a way appropriate for the specific program. To best understand how this happens, we need to delve into a simplified version of a component. All of the Java components work similarly.

Understanding Events

For concreteness, let's consider `JTextField`. A simplified version appears in Listing 13-8. The key feature is the `handleEvent` method. It detects various kinds of **events** caused by the user, such as pressing the Enter key or using the Tab key to move either into or out of the text field. Listing 13-8 uses pseudocode for detecting these actions because we don't really need to know how they are accomplished. Thanks to encapsulation and information hiding, we can use the class without knowing those intimate details.

What is important is that when one of these events occurs, two things happen. First, the component constructs an **event object** describing the event and containing such information as when the event occurred, if any keys were pressed at the time, and which component created it.

Second, the component calls a specific method, passing the event object as an argument. This method is one that we write as part of our controller. It's in this method that we have an opportunity to take actions specific to our program, such as calling the `removeTokens` method in the model.

Listing 13-8: *A simplified version of* `JTextField`

```
1  public class JTextField extends ...
2  { private ActionListener actionListener;
3    private FocusListener focusListener;
4
5    public void addActionListener(ActionListener aListener)
6    { this.actionListener = aListener;
7    }
8
9    public void addFocusListener(FocusListener fListener)
10   { this.focusListener = fListener;
11   }
12
13   private void handleEvent()
14   { if (user pressed the "Enter" key)
```

Listing 13-8: *A simplified version of* `JTextField` (continued)

```
15        { construct an object, event, describing what happened
16          this.actionListener.actionPerformed(event);
17        } else if (user tabbed out of this text field)
18        { construct an object, event, describing what happened
19          this.focusListener.focusLost(event);
20        } else if (user tabbed into this text field)
21        { construct an object, event, describing what happened
22          this.focusListener.focusGained(event);
23        } else
24          ...
25      }
26  }
```

KEY IDEA

In Java, controllers implement methods defined in interfaces with names ending in `Listener`.

Obviously, the method called has a name. That means that our controller must have a method with the same name. Ensuring that it does is a perfect job for a Java interface. The names `ActionListener` and `FocusListener` at lines 2, 3, 5, and 9 in Listing 13-8 are, in fact, the names of Java interfaces. Our controllers will always implement at least one interface whose name ends with `Listener`.

KEY IDEA

In Java, we use listener interfaces to implement controllers.

There are, unfortunately, two competing terminologies. "Controller" is a well-established name for the part of a user interface that interprets events and calls the appropriate commands in the model. Java uses the term **listener** for a class that is called when an event occurs. Most of the time the two terms mean the same thing.

Implementing a Controller

When the user presses the Enter key inside a `JTextField` component, the component calls a method named `actionPerformed`. This method is defined in the `ActionListener` interface (and is, in fact, the only method defined there). It takes a single argument of type `ActionEvent`. Therefore, the skeleton for our controller class will be:

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class RemovesController extends Object
                    implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
  }
}
```

Inside `actionPerformed`, we need to obtain the value the user typed into the text field and then call the model with that value. One approach is to have instance variables storing references to the text field and the model for the game. Then `actionPerformed` can be written as

```
public void actionPerformed(ActionEvent e)
{ String enteredText = this.textfield.getText();
  int remove = convert enteredText to an integer;
  this.model.removeTokens(remove);
}
```

The conversion from a string to an integer can be done with `parseInt`, a static method in the `Integer` class. It will throw a `NumberFormatException` if the user enters text that is not a valid integer. If this exception is thrown, we'll recover in the `catch` clause by selecting the entered text and ignoring what was entered.

The full method is shown in lines 21–29 of Listing 13-9. The rest of the listing, lines 11–19, is simply declaring the instance variables needed and initializing them in a constructor.

LOOKING AHEAD

Implementing controllers can use a number of shortcuts. Some of them will be explored in Section 13.6, Controller Variations.

Listing 13-9: *A controller for a text field*

```
1  import javax.swing.JTextField;
2  import java.awt.event.*;
3
4  /** A controller for the game of Nim that informs the model how many tokens a player
5  *  wants to remove.
6  *
7  *  @author Byron Weber Becker */
8  public class RemovesController extends Object
9                implements ActionListener
10 {
11   private NimModel model;
12   private JTextField textfield;
13
14   public RemovesController(NimModel aModel,
15                JTextField aTextfield)
16   { super();
17     this.model = aModel;
18     this.textfield = aTextfield;
19   }
20
21   public void actionPerformed(ActionEvent e)
22   { try
23     { int remove =
24                Integer.parseInt(this.textfield.getText());
25       this.model.removeTokens(remove);
```

FIND THE CODE

ch13/nimOneView/

Listing 13-9: *A controller for a text field* (continued)

```
26      } catch (NumberFormatException ex)
27      { this.textfield.selectAll();
28      }
29    }
30  }
```

Registering Controllers

The very last step to make this user interface interactive is to construct the controllers and register them with the text fields. Recall that the simplified version of `JTextField` shown in Listing 13-8 contained methods such as `addActionListener` and `addFocusListener`. They each took an instance of the similarly named interface and saved it in an instance variable. **Registering** our controller simply means calling the appropriate `add`*`Xxx`*`Listener` method for the relevant component, passing an instance of the controller as an argument.

KEY IDEA

A controller must be registered with a component.

We've only written one controller class, but we'll use one instance of it for the `redRemoves` text field and a second instance for the `blackRemoves` text field. A user interface often has several controllers, so it makes sense to have a helper method, `registerControllers`, just for constructing and registering controllers. It is called from the view's constructor.

The code in Listing 13-10 registers the red controller in two steps but combines the steps for the black controller.

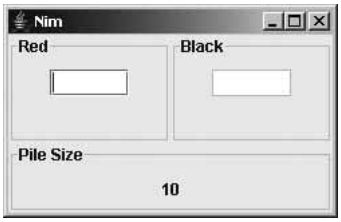
Listing 13-10: *A method registering the controllers with the appropriate components*

```
1  public class NimView extends JPanel implements IView
2  { // Instance variables omitted.
3
4    public NimView()
5    { // Some details omitted.
6      this.registerControllers();
7    }
8
9    /** Register controllers for the components the user can manipulate. */
10   private void registerControllers()
11   { RemoveController redController =
12         new RemoveController(this.model, this.redRemoves);
13     this.redRemoves.addActionListener(redController);
```

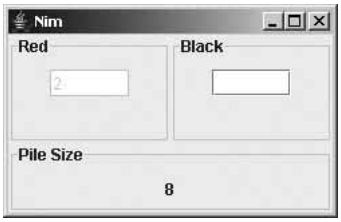
Listing 13-10: *A method registering the controllers with the appropriate components* (continued)

```
14
15      this.blackRemoves.addActionListener(
16         new RemoveController(this.model, this.blackRemoves));
17    }
18  }
```

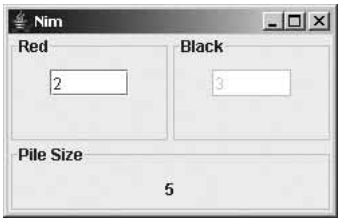
If you run the program with these additions, you should be able to play a complete, legal game, as shown in Figure 13-8.



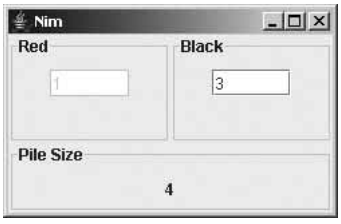
a) The game begins with a pile of 10. Red has the first turn.



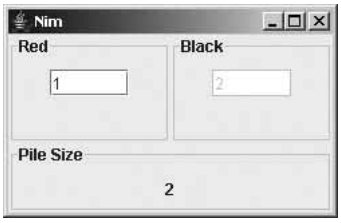
b) Red takes two tokens; now it's black's turn. The player must click in its text field before entering a value.



c) Black takes three tokens. It's red's turn. The "2" from red's previous turn still shows. Red does *not* need to click in its text field before entering a value but must delete the old value before entering a new one.



d) Red takes one token; now it's black's turn. The 3 from the previous turn still shows in the text field.



e) Black takes two tokens, setting up red for a win.

f) Red takes two tokens and is proclaimed the winner.

(figure 13-8)

User interface as it appears at each stage of a complete game

13.4.5 Refining the View

The program runs, as shown in Figure 13-8. However, there are three areas in which improvements could be made.

- The black user must click in its text field before entering a value. It would be nice if the player could simply type a new value.
- The value previously entered by a player remains in the text field and must be removed before entering a new value.
- Finally, the fonts used in the text fields and the `JLabel`s are too small, given their importance in the user interface.

Focus

In any given user interface, one component at most will receive input from the user's keyboard. This component is said to have the **keyboard focus**. Usually a component will give some visible sign when it has the focus. A component that accepts text will show a flashing bar called the **insertion point**. A button that has the focus will often have a subtle box around its label.

Focus normally shifts from one component to the next in the order that they were added to their container. In the case of Nim, however, the component that should have the focus depends on whose turn it is. So, in the `updateView` method, we can update which component has the focus with the following code. This code also replaces the previously entered value with an empty string.

```
if (this.model.getWhoseTurn() == Player.RED)
{ this.redRemoves.requestFocusInWindow();
  this.redRemoves.setText("");
} else if (this.model.getWhoseTurn() == Player.BLACK)
{ this.blackRemoves.requestFocusInWindow();
  this.blackRemoves.setText("");
}
```

Another approach is to write a controller class implementing the `FocusListener` interface. It can detect when a component gains or loses focus. This is useful, for example, if action needs to be taken when a user moves into or out of a component using either the mouse or the keyboard.

Fonts

A larger font for the various components can be specified with the `setFont` method. Its argument is a `Font` object describing the desired font. The following code could be included in the `layoutView` method to change the font for the five components.

```
// Enlarge the fonts.
Font font = new Font("Serif", Font.PLAIN, 24);
this.redRemoves.setFont(font);
this.blackRemoves.setFont(font);
this.redWins.setFont(font);
this.blackWins.setFont(font);
this.pileSize.setFont(font);
```

The first argument to the `Font` constructor specifies to use a font with **serifs**. Such fonts have short lines at the ends of the main strokes of each letter. Common fonts that have serifs include Times New Roman, Bookman, and Palatino. The string "SansSerif" can be used to specify a font without serifs. Helvetica is a common sans serif font. The string "monospaced" indicates a font using a fixed width for each letter. An example is Courier.

You can also specify an actual font name like "Helvetica" as the first argument. However, you can't be sure that the font is actually installed on the computer unless you check. The program in Listing 13-11 will list all the names of all the fonts that are installed. Try it for yourself to see which fonts are installed on your computer.

Listing 13-11: *A program to list the names of fonts installed on a computer*

```
1  import java.awt.Font;
2  import java.awt.GraphicsEnvironment;
3
4  /** List the font names available on the current computer system.
5  *
6  *  @author Byron Weber Becker */
7  public class ListFonts extends Object
8  { public static void main(String[] args)
9    { GraphicsEnvironment ge =
10         GraphicsEnvironment.getLocalGraphicsEnvironment();
11     Font[] names = ge.getAllFonts();
12
13     for (Font f : names)
14     { System.out.println(f.getName());
15     }
16   }
17 }
```

FIND THE CODE

ch13/fonts/

The second argument to the `Font` constructor is the style. There are three basic styles, defined as constants in the `Font` class: `PLAIN`, `ITALIC`, and `BOLD`. `ITALIC` makes the letters slant and `BOLD` makes the strokes thicker. A bold, italic font can also be specified by adding the `BOLD` and `ITALIC` constants together and passing the result to the constructor.

The third argument to the `Font` constructor is the font's size. The size is measured in **points**, where one point is 1/72 of an inch. Ten to 12 points is a comfortable size for reading; use 16 points or larger for labels and headlines.

This finishes our first view. The complete code is shown in Listing 13-12. Most components have many other ways to refine the way they look. Investigating them further falls outside the scope of this book. Exploring the documentation and method names for the component, as well as its superclasses, will often indicate what can be done.

FIND THE CODE

ch13/nimOneView/

Listing 13-12: *The completed code for the* `NimView` *class*

```
1  import javax.swing.JPanel;
2  import becker.util.IView;
3  import javax.swing.JTextField;
4  import javax.swing.JLabel;
5  import javax.swing.BorderFactory;
6  import java.awt.GridLayout;
7  import java.awt.BorderLayout;
8  import java.awt.Font;
9
10 /** Provide a view of the game of Nim to a user.
11 *
12 *  @author Byron Weber Becker */
13 public class NimView extends JPanel implements IView
14 {  // The model implementing Nim's logic.
15    private NimModel model;
16
17    // Get how many tokens to remove.
18    private JTextField redRemoves = new JTextField(5);
19    private JTextField blackRemoves = new JTextField(5);
20
21    // Info to display.
22    private JLabel pileSize = new JLabel();
23    private JLabel redWins = new JLabel("Winner!");
24    private JLabel blackWins = new JLabel("Winner!");
25
26    /** Construct the view.
27    *  @param aModel The model we will be displaying. */
28    public NimView(NimModel aModel)
29    { super();
30      this.model = aModel;
31
32      this.layoutView();
33      this.registerControllers();
34
```

Listing 13-12: *The completed code for the* `NimView` *class* (continued)

```
35      this.model.addView(this);
36      this.updateView();
37    }
38
39    /** Called by the model when it changes. Update the information this view displays. */
40    public void updateView()
41    { this.pileSize.setText("" + this.model.getPileSize());
42
43      this.redRemoves.setEnabled(
44            this.model.getWhoseTurn() == Player.RED);
45      this.blackRemoves.setEnabled(
46            this.model.getWhoseTurn() == Player.BLACK);
47      this.redWins.setVisible(
48            this.model.getWinner() == Player.RED);
49      this.blackWins.setVisible(
50            this.model.getWinner() == Player.BLACK);
51
52      if (this.model.getWhoseTurn() == Player.RED)
53      { this.redRemoves.requestFocusInWindow();
54        this.redRemoves.setText("");
55      } else if (this.model.getWhoseTurn() == Player.BLACK)
56      { this.blackRemoves.requestFocusInWindow();
57        this.blackRemoves.setText("");
58      }
59    }
60
61    /** Layout the view. */
62    private void layoutView()
63    { // A panel for the red player
64      JPanel red = new JPanel();
65      red.add(this.redRemoves);
66      red.add(this.redWins);
67      red.setBorder(BorderFactory.createTitledBorder("Red"));
68
69      // A panel for the black player
70      JPanel black = new JPanel();
71      black.add(this.blackRemoves);
72      black.add(this.blackWins);
73      black.setBorder(BorderFactory.createTitledBorder("Black"));
74
75      // Pilesize info.
76      JPanel pSize = new JPanel();
```

Listing 13-12: *The completed code for the* `NimView` *class* (continued)

```
77      pSize.add(this.pileSize);
78      pSize.setBorder(
79            BorderFactory.createTitledBorder("Pile Size"));
80
81      // Group the red and black panels.
82      JPanel center = new JPanel();
83      center.setLayout(new GridLayout(1, 2));
84      center.add(red);
85      center.add(black);
86
87      // Lay out the pieces in this view.
88      this.setLayout(new BorderLayout());
89      this.add(center, BorderLayout.CENTER);
90      this.add(pSize, BorderLayout.SOUTH);
91
92      // Enlarge the fonts.
93      Font font = new Font("Serif", Font.PLAIN, 24);
94      this.redRemoves.setFont(font);
95      this.blackRemoves.setFont(font);
96      this.redWins.setFont(font);
97      this.blackWins.setFont(font);
98      this.pileSize.setFont(font);
99    }
100
101   /** Register controllers for the components the user can manipulate. */
102   private void registerControllers()
103   { this.redRemoves.addActionListener(
104         new RemovesController(this.model, this.redRemoves));
105     this.blackRemoves.addActionListener(
106       new RemovesController(this.model, this.blackRemoves));
107   }
108 }
```

13.4.6 View Pattern

Views can be complex. However, they follow a common pattern, shown in Listing 13-13, which makes them much easier to understand and implement.

Listing 13-13: *A pattern template for a view*

```
1  import becker.util.IView;
2  import javax.swing.JPanel;
3  «list of other imports»
4
5  public class «viewName» extends JPanel implements IView
6  { private «modelClassName» model;
7
8    «component declarations»
9
10   public «viewName»(«modelClassName» aModel)
11   { super();
12     this.model = aModel;
13     this.layoutView();
14     this.registerControllers();
15     this.model.addView(this);
16     this.updateView();
17   }
18
19   public void updateView()
20   { «statements to update the components in the view»
21   }
22
23   private void layoutView()
24   { «statements to lay out the components within the view»
25   }
26
27   private void registerControllers()
28   { «statements to construct and register controllers»
29   }
30 }
```

13.5 Using Multiple Views

Now let's implement a different user interface for the same game. Because the `NimModel` class exhibits very low coupling with its first view (calling only the `updateView` method via the `IView` interface), we will be able to replace the user interface without changing `NimModel` at all.

KEY IDEA

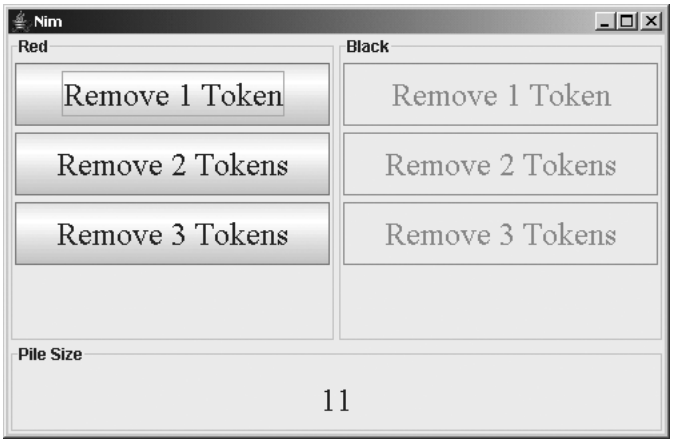
One of the strengths of the Model-View-Controller pattern is the low coupling between the various parts.

Our new interface is illustrated in Figure 13-9. Instead of typing in the number of tokens to remove, the user clicks the appropriate button. Like our previous interface,

components are disabled when they don't apply. For example, the black player's buttons are shown disabled, and when there are only 2 tokens remaining on the pile, the "Remove 3 Tokens" button will be disabled for both players. Like our previous interface, "Winner!" is displayed for the winning player at the appropriate time.

(figure 13-9)

Different user interface for Nim



We could write this user interface as one big view, as we did previously. However, this view has a total of nine components to manage, raising the overall complexity. Furthermore, the four components for the red player are managed almost exactly like those for the black player. This suggests that some good abstractions might simplify the problem.

KEY IDEA

A view can be partitioned into subviews.

Recall that we wrote the model anticipating multiple views. The model has a list of views, and each time the model's state changes, it goes through that list and tells each view to update itself. This allows us to decompose the overall view into a number of subviews. Each subview will add itself to the model's list of views and will have its `updateView` method called at the appropriate times.

This version of the interface will use three subviews: one for the red player, one for the black player, and one to display the pile size. `NimView` will still exist to organize the three subviews.

Dividing the view into several subviews has two distinct advantages. First, each view can focus on a smaller part of the overall job, allowing it to be simpler, easier to understand, easier to write, and easier to debug. Second, subviews can be easily changed or even replaced without fear of breaking the rest of the interface.

13.5.1 Implementing `NimView`

`NimView` is the overall view of the game. It is composed of the three subviews for the players and the pile size. `NimView` does not (directly) display information about the model nor does it (directly) update the model. Both of those tasks are delegated to the subviews. `NimView`'s only task is to organize the subviews in a panel.

In the following ways, it is a degenerate view:

- It doesn't need an instance variable storing a reference to the model.
- It doesn't have any controllers to construct or register.
- It doesn't need to register itself with the model.

As seen in Listing 13-14, all `NimView` does is instantiate and lay out the subviews.

Listing 13-14: `NimView`, *a view consisting of three subviews*

FIND THE CODE

ch13/nimMultiView/

```
1  import javax.swing.JPanel;
2  import javax.swing.BorderFactory;
3  import java.awt.GridLayout;
4  import java.awt.BorderLayout;
5
6  /** Provide a view of the game of Nim to a user.
7  *
8  *  @author Byron Weber Becker */
9  public class NimView extends JPanel
10 {
11    /** Construct the view.
12    *  @param aModel The model we will be displaying. */
13    public NimView(NimModel aModel)
14    { super();
15
16       // Create the subviews.
17       NimPlayerView red =
18             new NimPlayerView(aModel, Player.RED);
19       NimPlayerView black =
20             new NimPlayerView(aModel, Player.BLACK);
21       NimPileView pile = new NimPileView(aModel);
22
23       // Put a title on each subview.
24       red.setBorder(BorderFactory.createTitledBorder("Red"));
25       black.setBorder(BorderFactory.createTitledBorder("Black"));
26       pile.setBorder(BorderFactory.createTitledBorder("Pile Size"));
27
```

Listing 13-14: `NimView`, *a view consisting of three subviews* (continued)

```
28        // Group the red and black views.
29        JPanel center = new JPanel();
30        center.setLayout(new GridLayout(2, 1));
31        center.add(red);
32        center.add(black);
33
34        // Lay out the pieces in this view.
35        this.setLayout(new BorderLayout());
36        this.add(center, BorderLayout.CENTER);
37        this.add(pile, BorderLayout.SOUTH);
38     }
39  }
```

13.5.2 Implementing `NimPileView`

The `NimPileView` class, shown in Listing 13-15, is a simple view. It does not need to update the model, so there are no controllers. It only has a `JLabel` that is updated via the `updateView` method. `addView` is called at line 17 to add this view to the model's list of views.

FIND THE CODE

ch13/nimMultiView/

Listing 13-15: *The* `NimPileView` *class*

```
1  import becker.util.IView;
2  import javax.swing.*;
3  import java.awt.Font;
4
5  /** A view showing the current pile size for the game of Nim.
6  *
7  *  @author Byron Weber Becker */
8  public class NimPileView extends JPanel implements IView
9  { private NimModel model;
10   private JLabel pileSize = new JLabel();
11
12   /** Construct the view. */
13   public NimPileView(NimModel aModel)
14   { super();
15     this.model = aModel;
16     this.layoutView();
17     this.model.addView(this);
18     this.updateView();
```

Listing 13-15: *The* `NimPileView` *class* (continued)

```
19    }
20
21    /** Update the view. Called by the model when its state changes. */
22    public void updateView()
23    { this.pileSize.setText("" + this.model.getPileSize());
24    }
25
26    /** Layout the view. */
27    private void layoutView()
28    { this.pileSize.setFont(new Font("Serif", Font.PLAIN, 24));
29      this.add(this.pileSize);
30    }
31 }
```

13.5.3 Implementing `NimPlayerView`

`NimPlayerView` is a full-fledged view. It has its own components to lay out within itself. Those components are used to update the model, so they need to have controllers registered. The view also displays part of the state of the model—who's turn it is and who has won—and so it needs an `updateView` method and an instance variable to store a reference to the model.

We'll write `NimPlayerView` so that one instance of the class can be used for the red player and a second instance for the black player. To meet this goal, it must store the player it represents (lines 14 and 29 of Listing 13-16). The player is used in the `updateView` method (lines 45 and 48) to determine which buttons to enable and whether a winner should be declared.

The view has three buttons for user interaction. They all need to be added to the view, be enabled and disabled as appropriate, and have controllers registered. These tasks are all made easier by placing the buttons in an array (lines 16–20) and using loops (lines 43–46, 58–61, and 69–72).

Listing 13-16: *The* `NimPlayerView` *class*

```
1  import becker.util.IView;
2  import javax.swing.JPanel;
3  import javax.swing.JButton;
4  import javax.swing.JLabel;
5  import javax.swing.SwingConstants;
```

FIND THE CODE

ch13/nimMultiView/

Listing 13-16: *The* `NimPlayerView` *class* (continued)

```
6  import java.awt.Font;
7  import java.awt.GridLayout;
8
9  /** Provide a view of the game of Nim focused on one particular player to a user.
10 *
11 *  @author Byron Weber Becker */
12 public class NimPlayerView extends JPanel implements IView
13 { private NimModel model;
14   private Player player;
15
16   private JButton[] removeButtons = new JButton[] {
17     new JButton("Remove 1 Token"),
18     new JButton("Remove 2 Tokens"),
19     new JButton("Remove 3 Tokens")
20   };
21   private JLabel winner = new JLabel("Winner!");
22
23   /** Construct a view for one player.
24    *  @param aModel    The game's model.
25    *  @param player    The player for which this is the view. */
26   public NimPlayerView(NimModel aModel, Player aPlayer)
27   { super();
28     this.model = aModel;
29     this.player = aPlayer;
30
31     this.layoutView();
32     this.registerControllers();
33
34     this.model.addView(this);
35     this.updateView();
36   }
37
38   /** Update the view to reflect recent changes in the model's state. */
39   public void updateView()
40   { Player whoseTurn = this.model.getWhoseTurn();
41     int pSize = this.model.getPileSize();
42     // Enable buttons if it's my player's turn and there are enough tokens on the pile.
43     for (int i = 0; i < this.removeButtons.length; i++)
44     { this.removeButtons[i].setEnabled(
45           whoseTurn == this.player && i + 1 <= pSize);
46     }
47     this.winner.setVisible(
48           this.model.getWinner() == this.player);
```

Listing 13-16: *The* `NimPlayerView` *class* (continued)

```
49    }
50
51    /** Lay out the components for this view. */
52    private void layoutView()
53    { GridLayout grid = new GridLayout(4, 1, 5, 5);
54      this.setLayout(grid);
55
56      Font font = new Font("Serif", Font.PLAIN, 24);
57
58      for (JButton b : this.removeButtons)
59      { this.add(b);
60        b.setFont(font);
61      }
62
63      this.winner.setFont(font);
64      this.add(this.winner);
65    }
66
67    /** Register controllers for this view's components. */
68    private void registerControllers()
69    { for (int i = 0; i < this.removeButtons.length; i++)
70      { this.removeButtons[i].addActionListener(
71            new RemoveButtonController(this.model, i + 1));
72      }
73    }
74  }
```

Like `JTextField`, `JButton` objects use an `ActionListener`. When the button is clicked, it calls the `actionPerformed` method for all the listeners that have been added. Recall that it is inside the `actionPerformed` method that we specify the code to execute when the button is clicked. This is where we call the `removeTokens` method in the model.

In our previous controller the user typed the number of tokens to remove from the pile. We need a different way to find out how many tokens to remove. One approach is to have a separate controller object for each button. The controller has an instance variable that remembers how many tokens to remove. That instance variable is set, of course, when the controller is constructed. We can see this at line 71 of Listing 13-16, where a new controller is instantiated for each button.

The revised controller class is shown in Listing 13-17.

FIND THE CODE

ch13/nimMultiView/

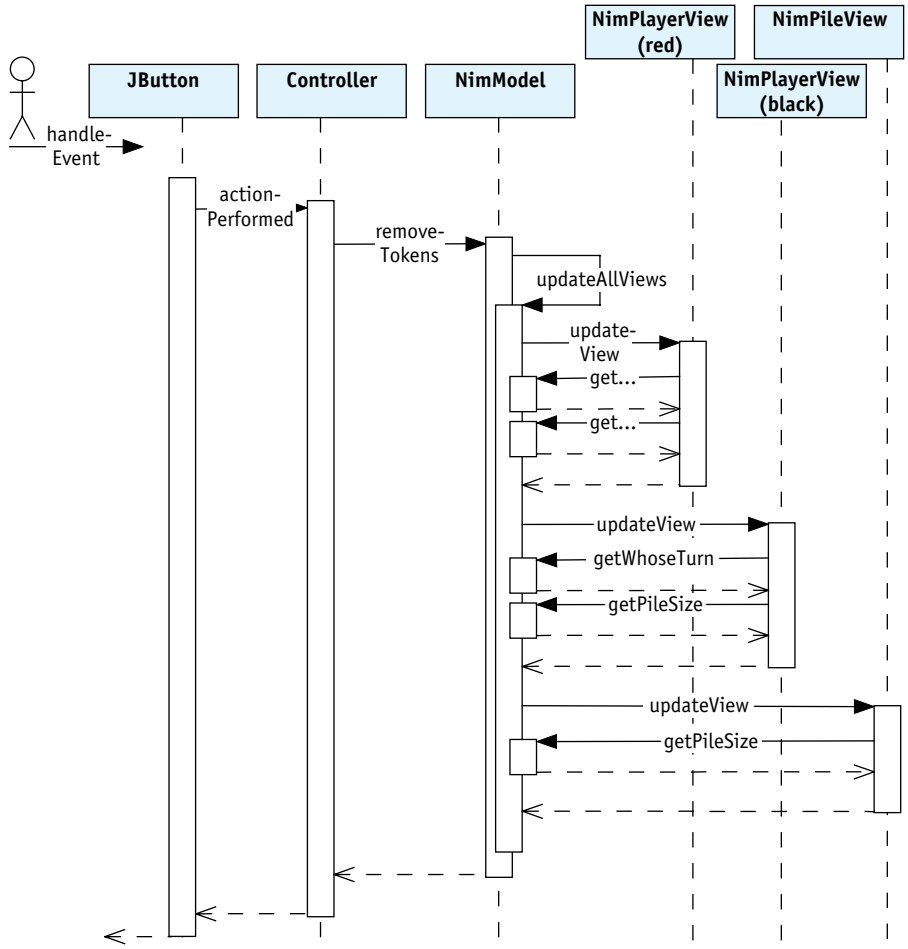
Listing 13-17: *The controller for the* `JButton`*s used to remove tokens*

```
1  import java.awt.event.*;
2
3  /** A controller to remove tokens from the game of Nim.
4  *
5  *  @author Byron Weber Becker */
6  public class RemoveButtonController extends Object
7                implements ActionListener
8  {
9    private NimModel model;
10   private int numRemove;
11
12   /** Construct an instance of the cotroller.
13   *  @param aModel       The model this controls.
14   *  @param howMany      How many tokens to remove when the button is clicked. */
15   public RemoveButtonController(NimModel aModel, int howMany)
16   { super();
17     this.model = aModel;
18     this.numRemove = howMany;
19   }
20
21   /** Remove the right number of tokens from the model. */
22   public void actionPerformed(ActionEvent evt)
23   { this.model.removeTokens(this.numRemove);
24   }
25 }
```

13.5.4 Sequence Diagrams

Removing a token involves six interacting classes. This is a level of complexity that we haven't seen before, but it is not uncommon. To keep things in perspective, it's important to think locally. For each method, we can ask, what is the job that this method has to do? What services does it need from other classes to do that job?

But a global perspective can help, too. Figure 13-10 is a **sequence diagram** that can help visualize the objects involved in removing a token and the sequence of actions taking place.



(figure 13-10)

Sequence diagram of the actions involved in removing tokens and updating the views

The six objects involved are shown at the top of the diagram, each with their class name. In the case of `NimPlayerView` there are two, so we distinguish between the instance for the red player and the instance for the black player. There are six `JButton` objects, but it isn't important to distinguish between them, so only one is shown.

The dashed line extending down from each object is its **lifeline**. In a complete sequence diagram, the lifeline would begin with the object's construction and end when the object is no longer needed. The boxes along the lifeline represent a method executing in that object. The solid arrows between the boxes represent one method calling another. A dashed arrow with an open arrowhead represents a method finishing execution and returning to its caller.

Putting all this together, the diagram begins in the upper-left corner with the `handleEvent` method in `JButton` being called, presumably because the user clicked

the button. `handleEvent` calls the `actionPerformed` method in the controller. We can think of the `actionPerformed` method as executing for quite a while—all the time that it takes to call `removeTokens`, including the calls that `removeTokens` makes. This length of time is represented by the length of the box on the controller's lifeline.

On the lifeline for `NimModel`, we see that the longest box, corresponding to `removeTokens`, calls a helper method in the same class, `updateAllViews`. This helper method calls all the `updateView` methods in the views registered with `NimModel`. Each of these, of course, calls additional methods.

By the time execution returns to the `handleEvent` method in `JButton` at the bottom-left corner of the diagram, tokens have been removed from the model and all of the views have been updated accordingly.

13.6 Controller Variations

Three techniques are often used to simplify writing controllers. One nests the controller class inside the view's class. The second makes use of information passed in the event objects. The third is a shortcut often taken in sample code in other books and on the Internet.

13.6.1 Using Inner Classes

An **inner class** is a class that is nested inside another class.[2] Inner classes are most useful for defining small helper classes that are very specific to a particular task. By placing inner classes inside the class they are helping, we can make that relationship more explicit and keep the definition of the helper class very close to the class it is helping. Beyond this, the primary advantage of an inner class is that it can access the methods and instance variables of its enclosing class—even the private methods and instance variables.

KEY IDEA

An inner class can access instance variables and methods of its enclosing class.

Views are usually written with inner classes for the controllers.

Listing 13-18 shows the `NimPlayerView` (Listing 13-16) and `RemoveButtonController` (Listing 13-17) combined in a single file by making the controller an inner class.

[2] There are actually four varieties of inner classes. We will focus on member classes. The other three are nested top-level classes, local classes, and anonymous classes.

The first thing to notice about Listing 13-18 is that `RemoveButtonController` falls between the opening and closing braces of the `NimPlayerView` class. The actual order of instance variables, methods, and inner classes within the outer class doesn't matter to the compiler, but inner classes are generally placed at the end.

KEY IDEA

An inner class is placed inside another class, but outside of all methods.

FIND THE CODE

ch13/nimInnerClass/

Listing 13-18: *Using an inner class for a view's controller*

```
1  // Import classes needed by both view and controller.
2  public class NimPlayerView extends JPanel implements IView
3  { private NimModel model;
4
5    // Other instance variables, constructor, updateView, and layoutView are omitted.
6
7    private void registerControllers()
8    { for (int i = 0; i < this.removeButtons.length; i++)
9      { this.removeButtons[i].addActionListener(
10           new RemoveButtonController(i+1));
11     }
12   }
13
14   // Inner class for the controllers to remove tokens from the pile.
15   private class RemoveButtonController extends Object
16                implements ActionListener
17   { private int numRemove;
18
19     public RemoveButtonController(int howMany)
20     { super();
21       this.numRemove = howMany;
22     }
23
24     public void actionPerformed(ActionEvent evt)
25     { NimPlayerView.this.model.removeTokens(this.numRemove);
26     }
27   }
28 }
```

Second, the inner class accesses the `model` instance variable from the outer class at line 25. The syntax for doing so is a little odd. We *cannot* write `this.model` because then we would be referring to an instance variable in the `RemoveButtonController` class. To access the outer class, first give the name of that class and then access the variable as usual. It is also possible to write the following and let the compiler figure it out:

```
model.removeTokens(this.numRemove);
```

For clarity, however, we will always write the longer version.

Third, because the inner class can access the model via the outer class, the `model` instance variable has disappeared along with code in the constructor to initialize it. The argument is also omitted when the constructor is called in line 10.

Each instance of the inner class is tied to a specific instance of the outer class. For example, the game creates two instances of `NimPlayerView`, one for the red player and one for the black player. Both of these objects create three controllers. The controllers created for red's instance of the view are forever tied to that instance. They will access the methods and instance variables in red's instance of the view and will never access those in black's instance.

13.6.2 Using Event Objects

The `actionPerformed` method is always passed an `ActionEvent` object which provides more details about the user's action. All of the methods in all of the listener interfaces have an event object as a parameter.

KEY IDEA

Use event objects to obtain more information about the event and the source that generated it.

One of the most useful items of information in an event object is the source of the event—that is, which component was manipulated by the user. Using that information, we can figure out how many tokens to remove without using an instance variable in the controller class. We'll simply compare the source to each `JButton` in the array. When we have a match, we'll know how many tokens to remove.

With this approach, the controller will have no instance variables at all. This has two implications. First, there are no instance variables to initialize, and we can let Java provide a default constructor for us.[3] Second, every instance is just like all the other instances, and we can use the same controller for all three buttons. Listing 13-19 shows how.

FIND THE CODE

ch13/nimInnerClass/

Listing 13-19: *A controller that uses the event object to avoid instance variables*

```
1  // Import classes needed by both view and controller.
2  public class NimPlayerView extends JPanel implements IView
3  { private NimModel model;
4    private JButton[] removeButtons = new JButton[]
5    { new JButton("Remove 1 Token"),
6      new JButton("Remove 2 Tokens"),
7      new JButton("Remove 3 Tokens")
8    };
9
10   // Other instance variables, constructor, updateView, and layoutView are omitted.
```

[3] Omitting the parameterless or default constructor is an option for every class, but we have always included it, when applicable, for clarity. Controllers are usually so small and specialized, however, that we can omit them without loss of clarity.

Listing 13-19: *A controller that uses the event object to avoid instance variables* (continued)

```
11
12    /** Register controllers for this view's components. */
13    private void registerControllers()
14    { RemoveButtonController controller =
15            new RemoveButtonController();
16      for (int i = 0; i < this.removeButtons.length; i++)
17      { this.removeButtons[i].addActionListener(controller);
18      }
19    }
20
21    private class RemoveButtonController extends Object
22                implements ActionListener
23    { public void actionPerformed(ActionEvent evt)
24      { JButton src = (JButton)evt.getSource();
25        if (src == removeButtons[0])
26        { model.removeTokens(1);
27        } else if (src == removeButtons[1])
28        { model.removeTokens(2);
29        } else if (src == removeButtons[2])
30        { model.removeTokens(3);
31        } else
32        { assert false;          // Shouldn't happen!
33        }
34      }
35    }
36  }
```

Note in line 24 that the `getSource` method returns an `Object` which must be cast to an appropriate type. The source itself will often have useful information. For example, if it were a text field, we could get the text typed by the user.

The cascading-`if` structure in lines 25–33 is fine for a small number of components, but if the components are stored in an array, a loop can be more concise, as follows:

```
public void actionPerformed(ActionEvent evt)
{ JButton src = (JButton)evt.getSource();
  int i = 0;
  while (removeButtons[i] != src)
  { i++;
  }
  assert removeButtons[i] == src;
  model.removeTokens(i+1);
}
```

13.6.3 Integrating the Controller and View

KEY IDEA

This is not a recommended approach, but its use is widespread.

The controller and view can also be integrated into the same class without the use of an inner class. Many examples on the Web use this approach because it is quick and easy. It introduces a significant disadvantage, however, in that there is only one controller for all of the various components. With the previous techniques, you can easily write one controller for a `JButton` and a different controller for a `JTextField`. Each controller has its own `actionPerformed` method that is specific to a particular task. When the controller and view are integrated, a single `actionPerformed` method must handle both components. In terms of the software engineering principles studied in Section 11.3.2, such integration reduces the cohesion of the methods (recall that we want high cohesion). Nevertheless, the technique is shown here so that you can understand it if and when you see it.

The technique works by implementing the required interfaces in the view class itself. In Listing 13-20, the `ActionListener` interface is listed on the class header (lines 2–3) and its only method, `actionPerformed`, is implemented at lines 14–23 just like any other method. Note that there is no inner class. The "controller" is registered with the `JButton` objects in line 10. Instead of constructing a separate object, a reference to the view itself (that is, `this`) is passed to the button.

FIND THE CODE

ch13/nimIntegrated/

Listing 13-20: *A version of* `NimPlayerView` *that integrates the view and the controller*

```
1  // Import classes needed by both view and controller.
2  public class NimPlayerView extends JPanel
3                 implements IView, ActionListener
4  {
5     // Other instance variables, constructor, updateView, and layoutView are omitted.
6
7     /** Register controllers for this view's components. */
8     private void registerControllers()
9     { for (int i = 0; i < this.removeButtons.length; i++)
10      { this.removeButtons[i].addActionListener(this);
11      }
12    }
13
14    public void actionPerformed(ActionEvent evt)
15    { JButton src = (JButton)evt.getSource();
16
17      int i = 0;
18      while (removeButtons[i] != src)
19      { i++;
20      }
21      assert removeButtons[i] == src;
```

Listing 13-20: *A version of* `NimPlayerView` *that integrates the view and the controller* (continued)

```
22        model.removeTokens(i+1);
23      }
24    }
```

13.7 Other Components

So far, we have only worked with `JTextField` and `JButton` components. But there are many more components, too many to cover in a book such as this. So how can you learn to use them? Use the following strategies:

- Discover what components are available and might be applicable
- Identify the listeners used
- Skim the documentation
- Begin with sample code
- Work incrementally

In the following sections, we'll use these strategies to learn how to display a set of color names to use in Nim instead of "Red" and "Black".

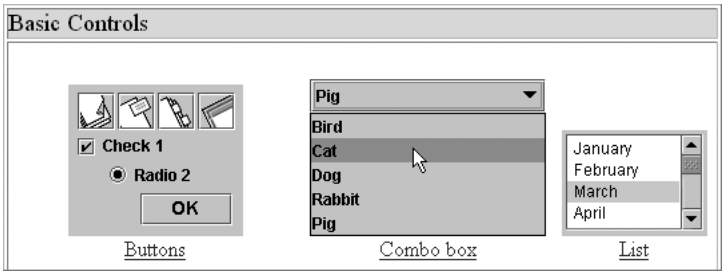
13.7.1 Discover Available Components

There are several ways to discover available components. One is to look at "A Visual Index to the Swing Components," which can be found at *http://java.sun.com/docs/books/tutorial/uiswing/components/components.html*. It shows a sample of each component and has links to documentation where you can learn more. Figure 13-11 shows a part of the Web page that looks promising. It appears that at least two kinds of components can display lists of color names, as we would like to do.

Clicking the links labeled "Combo box" and "List" leads to pages titled "How to Use Combo Boxes" and "How to Use Lists." The first page refers to the component `JComboBox`, and the second page refers to `JList`.

(figure 13-11)

Part of "A Visual Index to the Swing Components"



Another option is to find one of several demonstration programs available. One that comes with this textbook is shown in Figure 13-5. If you have downloaded the example code for the textbook, you'll find the code in the directory `ch13/componentDemo`. Running the program and playing with the components will show that `JSpinner` is also a possibility. In Figure 13-5, it displays "Sunday," but it also "spins" through the other days of the week. It could also spin through the color names we want to display.

Any of these options could work for us. Choosing between them is largely a matter of personal taste. For now, we'll choose `JList`.

13.7.2 Identify Listeners

When we identify the listeners for a component, we identify what kind of events it can tell us about and therefore what kind of controllers we can write. Every component may have the following six kinds of listeners:

- Component listeners listen for changes in the component's size, position, or visibility. Component listeners have methods like `componentHidden`, `componentResized`, and `componentMoved`.
- Focus listeners listen for the component gaining or losing the ability to receive keyboard input. Focus listeners have two methods, `focusGained` and `focusLost`.
- Key listeners listen for key press events. Such events are fired only by the component that has the keyboard focus. Key listeners have `keyPressed`, `keyReleased`, and `keyTyped` methods.
- Mouse listeners listen for mouse clicks and the mouse moving into and out of the component's drawing area. Mouse listeners have five methods, including `mouseEntered` and `mouseClicked`.
- Mouse motion listeners listen for changes in the cursor's position within the component. Such listeners have two methods, `mouseMoved` and `mouseDragged`.
- Mouse wheel listeners listen for mouse wheel movement over the component. They have a single method, `mouseWheelMoved`.

In addition to these six listeners, components have one or more additional listeners that vary by component type. For example, we have already seen that `JTextField` and `JButton` objects can have `ActionListeners`.

A complete table of components and listeners is maintained by the creators of Java at *http://java.sun.com/docs/books/tutorial/uiswing/events/eventsandcomponents.html*. This table is summarized in Figure 13-12. Looking at the list, we can tell that the `JList` component uses a `ListSelectionListener` and one or more unspecified listeners.

(figure 13-12)

Listeners used by some of Java's GUI components

Component	ActionListener	CaretListener	ChangeListener	DocumentListener	ItemListener	ListSelectionListener	WindowListener	Other
JButton	√		√		√			
JCheckBox	√		√		√			
JColorChooser			√					
JComboBox	√				√			
JDialog							√	
JEditorPane		√		√				√
JFileChooser	√							
JFormattedTextField	√	√		√				
JFrame							√	
JList						√		√
JMenu								√
JMenuItem	√		√		√			√
JPasswordField	√	√		√				
JPopupMenu								√
JProgressBar			√					
JRadioButton	√		√		√			
JSlider			√					
JSpinner			√					
JTabbedPane			√					
JTable						√		√
JTextArea		√		√				
JTextField	√	√		√				
JToggleButton	√		√		√			
JTree								√

Another approach is to look at the documentation for the component at *http://java.sun.com/j2se/1.5.0/docs/api/*. For example, find `JList` in the left side and click on it. Scroll down to the list of methods and look for methods named `addXxxxListener`, where the *`Xxxx`* can vary. `JList` has an `addListSelectionListener` method.

The documentation for `ListSelectionListener` says the interface specifies a single method, `valueChanged`. This is the method that our controller for `JList` will need to implement.

13.7.3 Skim the Documentation

There are two primary sources of information for working with Java's GUI components: the API documentation and the Java Tutorial.

Application Programming Interface (API) Documentation

KEY IDEA

The API documentation provides full details about each class.

One primary source of information is the **API**, or **application programming interface**, documentation. It is the class-by-class documentation found at *http://java.sun.com/j2se/1.5.0/docs/api/*. The documentation for each class gives an overview of the class, its inheritance hierarchy, a list of the constructors provided, and a list of the methods provided, including detailed descriptions of what they do.

The first time you use a component, skim this documentation looking for methods that sound useful. There may be lots of them—don't get overwhelmed. For `JList`, the documentation lists about 70 methods, plus the 344 methods it inherits from its superclasses.

What's important when getting started using a `JList`? Constructing the component, adding items to display in the list, adding a listener, and finding out which item on the list was selected. Skimming the documentation for methods that sound relevant yields the following:

- `JList()`: constructs an empty `JList`
- `JList(Object[] listData)`: constructs a `JList` that displays the elements in the specified array
- `addListSelectionListener`: adds a listener to the `JList`
- `getSelectedIndex`: returns the index of the first selected item; if nothing is selected, it returns -1
- `getSelectedIndicies`: returns an array of all the selected indices
- `getSelectedValue`: returns the first selected value

These methods answer most of our questions. We might have expected to find an "add item" method to add items to the list, but we didn't. Instead, it appears that we pass an array of items to display when the component is constructed. It also appears that several items can be selected at one time. We may want to make note of that for future reference.

The Java Tutorial

KEY IDEA

The Java Tutorial *contains lots of sample code.*

The *Java Tutorial* at *http://java.sun.com/docs/books/tutorial/* provides a wealth of practical examples for creating graphical user interfaces. Particularly relevant is the "Creating a GUI with JFC/Swing" chapter. It contains sections such as "Learning Swing by Example," "Using Swing Components," and "Writing Event Listeners." One subsection, at *http://java.sun.com/docs/books/tutorial/uiswing/components/componentlist.html*, contains a long list of topics with names like "How to Make Applets" and "How to Use Lists." The API documentation often provides direct links to these sections of the tutorial.

Clicking the "How to Use Lists" link opens a document that includes sample code and sections titled "Initializing a List," "Selecting Items in a List," and "Adding Items to and Removing Items from a List." All sound helpful!

13.7.4 Begin with Sample Code

Building on the discoveries of someone else is always easier than starting from scratch. When learning to use a new component, look for sample code using it. The *Java Tutorial* is a good place to look, particularly in the "How to…" sections referenced earlier.

Another source for sample code that matches the style presented in this textbook is the `componentDemo` program shown in Figure 13-5. If you run the program and click an element in the `JList`, an entry is added to the table at the bottom of the frame. The view column says "ListView." This is the name of the class containing the `JList`. The second column, "Listener," says "ListView$ListController." That's the name of the controller class that handled your mouse click—the `ListController` class that is an inner class within the `ListView` class.

Open the source for `ListView` and you'll find the code constructing the `JList`, laying it out within a view, and registering a controller, as well as the code for the controller itself. Much of this code can be cut and pasted directly into the program you're writing.

13.7.5 Work Incrementally

The last piece of advice is to work incrementally. Start with small goals for the component. Meet those goals and then move on to more ambitious goals. For example, you might begin by displaying the `JList` in a view. Listing 13-21 shows a minimal view with the goal of showing a `JList` with the names of some colors and detecting when one has been selected.

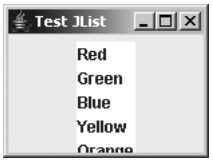
FIND THE CODE

ch13/usingJList/

Listing 13-21: *A simple view to display a list of colors and detect when one is selected*

```
1  import becker.util.IView;
2  import javax.swing.JPanel;
3  import javax.swing.JList;
4  import javax.swing.event.ListSelectionEvent;
5  import javax.swing.event.ListSelectionListener;
6
7  public class View extends JPanel implements IView
8  {  // private Object model;
9     private JList list;
10
11    public View(Object aModel)
12    {  super();
13       // this.model = aModel;
14       this.layoutView();
15       this.registerControllers();
16       // this.model.addView(this);
17       this.updateView();
18    }
19
20    public void updateView()
21    {// Statements to update the components in the view.
22    }
23
24    private void layoutView()
25    {  this.list = new JList(new String[] {"Red", "Green", "Blue",
26             "Yellow", "Orange", "Pink", "Black"});
27       this.add(this.list);
28    }
29
30    private void registerControllers()
31    {  this.list.addListSelectionListener(
32             new ListController());
33    }
34
35    private class ListController extends Object
36             implements ListSelectionListener
37    {  public void valueChanged(ListSelectionEvent evt)
38       {  System.out.println(
39             "selected " + View.this.list.getSelectedValue());
40       }
41    }
42 }
```

Running a program that places this view in a frame appears as shown in Figure 13-13 and proves that we have made significant progress. The list shows the seven colors and it prints a message when one is selected. However, there are two problems. First, each time a color is selected, two copies of the message are printed by the controller. Second, the list has no scroll bars. If the window is made smaller than the list, part of the list simply disappears.



(figure 13-13)

Running the `JList` *test*

For the first problem, it seems like the `ListSelectionListener` documentation would be a good place to start. After all, the listener contains the code that is being called twice. However, that documentation provides no help.

If we look at the `ListSelectionEvent` documentation, we find a method named `getValueIsAdjusting`. Its description says "Returns `true` if this is one of multiple change events," which sounds promising. `JList` reports a list selection event both when the mouse is pressed and when it is released—as well as several more events in between if the user moves the mouse over different values in the list. Rewriting our controller's `valueChanged` method results in only one message being printed, the one selected when the mouse button is released:

```
public void valueChanged(ListSelectionEvent evt)
{ if (!evt.getValueIsAdjusting())
  { System.out.println(
        "selected " + View.this.list.getSelectedValue());
  }
}
```

The problem of the missing scroll bars can be solved by searching the `JList` class documentation for "scroll." That search finds the following:

> "`JList` doesn't support scrolling directly. To create a scrolling list you make the `JList` the viewport view of a `JScrollPane`. For example:
>
> ```
> JScrollPane scrollPane = new JScrollPane(dataList);
> ```
>
> where `dataList` is the instance of `JList` you want to display. The `JScrollPane` component is added to the view instead of the `JList`."

Working incrementally, we add equivalent code to the `layoutView` method in Listing 13-21 and run the program to see the results. Unfortunately, nothing has changed, and scroll bars still do not appear.

KEY IDEA

Component-size problems are often related to the layout manager.

It turns out that `JPanel`'s default layout manager, `FlowLayout`, allows the list to take up as much space as it requests. `JScrollPane` does not show the scroll bars until the available space is less than the requested space. `BorderLayout` is a layout manager that forces its components to fit within the available space. Using it to manage the view's layout results in the scroll bars appearing when the `JList` is small. The resulting code for `layoutView` is as follows:

```
private void layoutView()
{ this.setLayout(new BorderLayout());
  this.list = new JList(new String[] {"Red", "Green",
    "Blue", "Yellow", "Orange", "Pink", "Black"});
  JScrollPane scrollpane = new JScrollPane(this.list);
  this.add(scrollpane, BorderLayout.CENTER);
}
```

As shown here, it is unrealistic to expect to understand and use a complex class like `JList` on the first try. An excellent strategy is to work incrementally. Understand and implement the basics, make note of the remaining issues, and then solve them one at a time. Using this strategy, we are well on our way to making effective use of the `JList` component. Reasonable next steps include making calls to the model in response to user selections and, if required, learning how to add new values to the list while the program is running.

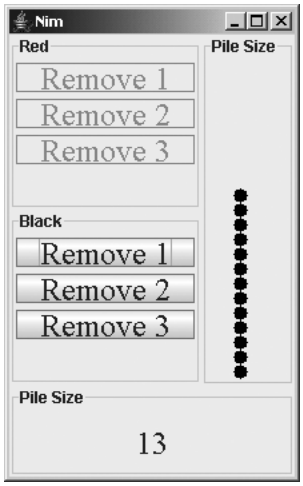
13.8 Graphical Views

Many components are available for Java programs, but sometimes none of them are quite right for a particular application. In those cases, you may need to make your own. We have, in fact, already done this. In Section 6.7, we wrote the `Thermometer` class, which displayed a temperature using an image of a thermometer.

In Section 13.8.1, we will implement a similar class to simply display a pile of tokens for the game of Nim. In Section 13.8.2, we will go a step further and add a listener for mouse events so that the user can utilize our new component to select the tokens to remove from the pile.

13.8.1 Painting a Component

Instances of our custom component, `PileComponent`, represent a pile of tokens as circles, drawn one on top of the other, as shown in Figure 13-14. Such a component that does its own painting usually extends the `JComponent` class (see Listing 13-22).



(figure 13-14)

Custom component representing a pile of tokens for Nim

Two crucial parts of the class are instance variables, used to either store or acquire the information required to do the painting (lines 4–5), and the `paintComponent` method (lines 27–40).

Two instance variables are required: `numTokens` stores the actual number of tokens to display; `maxTokens` stores the maximum number that could be displayed. The maximum is used to scale the circles appropriately; it is set with the constructor. `numTokens` is set using a mutator method, `setPileSize`, called from the `updateView` method in the view that contains the `PileComponent` object. When the pile size is changed, `this.repaint()` must be called. It tells the Java system that it should call `paintComponent` as soon as possible to redraw the pile.

LOOKING BACK

Repainting is explained in more detail in Section 6.7.2.

The `paintComponent` method begins by calculating useful values for painting (lines 29–32). The first two merely make temporary copies of the component's width and height to make them easier to use. The second two calculate the diameter of each token and where the left side will be painted.

Lines 35-39 use a loop to draw each of the tokens in the pile.

One other detail is setting the minimum and preferred size of the component in lines 12 and 13. Without these statements, the component's size will default to a barely visible 1 x 1 pixel square.

FIND THE CODE

ch13/nimMultiView/

Listing 13-22: *A component that displays the size of a token pile graphically*

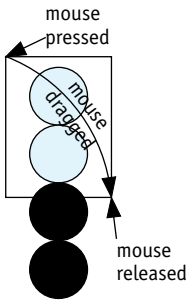
```
1  // Import statements omitted.
2  public class PileComponent extends JComponent
3  {
4    private int numTokens = 0;
5    private int maxTokens;
6
7    /** Create a new component.
8    *  @param max The maximum number of tokens that can be displayed. */
9    public PileComponent(int max)
10   { super();
11     this.maxTokens = max;
12     this.setMinimumSize(new Dimension(40, 60));
13     this.setPreferredSize(new Dimension(60, 90));
14   }
15
16   /**  Reset the size of the pile.
17   *  @param num The new pile size. 0 <= num <= maxTokens */
18   public void setPileSize(int num)
19   { if (num < 0 || num > this.maxTokens)
20     { throw new IllegalArgumentException("too many/few tokens");
21     }
22     this.numTokens = num;
23     this.repaint();
24   }
25
26   /** Paint the component. */
27   public void paintComponent(Graphics g)
28   { // Values to use in painting.
29     int width = this.getWidth();
30     int height = this.getHeight();
31     int tokenDia = Math.min(width, height/this.maxTokens);
32     int tokenLeft = width/2 - tokenDia;
33
34     // Draw the tokens.
35     g.setColor(Color.BLACK);
36     for (int i = 0; i < this.numTokens; i++)
37     { int top = height - (i + 1) * tokenDia;
38       g.fillOval(tokenLeft, top, tokenDia, tokenDia);
39     }
40   }
41 }
```

13.8.2 Making a Graphical Component Interactive

We can make `PileComponent` interactive, enabling users to use the mouse to select a number of tokens by performing the following steps, also illustrated in Figure 13-15.

The steps are:

- Press the mouse button
- Drag the mouse over some of the tokens displayed by the component
- Release the mouse button



(figure 13-15)

Sequence of mouse actions triggering a selection

In general, implementing a custom component involves the five steps shown in Figure 13-16. The result is a component we can use in a view, complete with its own controllers—just like we use controllers with `JTextField` and `JButton` components.

1. Write a class that extends `JComponent`.
2. Declare instance variables to store the information required to paint the component appropriately. Override the `paintComponent` method to do the painting.
3. Write mutator methods to update the instance variables. Call the `repaint` method before exiting any method that changes the component's state.
4. Declare a list to store the component's listeners. Include methods to add and remove components from the list, and a `handleEvent` method to inform all listeners of a significant event.
5. Write and register listeners to detect and respond to the user's actions.

(figure 13-16)

Steps to implement an interactive component

You may notice similarities with what we have done before. For example, both a component and a model call a method when their state is changed (Step 3), and both have a list of objects to inform when something significant happens (Step 4).

KEY IDEA

A component has features in common with both models and views.

On the other hand, a component is also similar to a view. Both extend a kind of component (`JPanel` versus `JComponent` in Step 1), and both have listeners (Step 5), although in a view the listeners are called "controllers."

The first three steps in Figure 13-16 were already done in the earlier version of `PileComponent`. In the following subsections, we will discuss Steps 4 and 5 in more detail, referring to Listing 13-23, which contains the code for the completed component. This new, interactive version of `PileComponent` will be called `PileComponent2`.

Informing the Component's Listeners

When our component is used in a view, we will want to add controllers to it that update the model. They will implement an interface such as `ActionListener` or `ListSelectionListener`. Now, because we are writing the component, we can choose which listener interface to use. Of all the listeners listed in Figure 13-12, `ActionListener` seems the most appropriate.

Therefore, in lines 22–23 of Listing 13-23 we declare an `ArrayList` to store objects implementing `ActionListener`. In lines 45–48 we provide an `addActionListener` method, like the one provided in `JButton` and `JTextField`. A complete implementation would also provide a `removeActionListener` method.

LOOKING BACK

Listing 13-8 shows a simplified version of `JTextField`*. It also has a* `handleEvent` *method.*

In lines 101–108 we provide a private method named `handleEvent`, to be called when the component detects the user selecting some tokens. It constructs an `ActionEvent` object and then loops through all the registered controllers, calling their `actionPerformed` method and passing the event object.

Writing and Registering Listeners

The last step, and the most complicated one, is figuring out when to call the `handleEvent` method. To do so, we will write two inner classes implementing `MouseListener` and `MouseMotionListener`. The first listener[4] will be informed each time something happens to the mouse button. The second listener will be informed each time the mouse moves. Mouse-related events are split into two listeners because there are *many* motion events. If the component only cares about mouse clicks, we don't want to incur the overhead associated with mouse motion events.

[4] We use the term "listener" rather than "controller" because these classes will not be interacting with the program's model.

We need to detect the following three mouse events:

- When the mouse button is pressed, we will create a new rectangle that will bound the area (and tokens) selected.
- When the mouse is dragged, we will update the size of the bounding rectangle and repaint the component to show it.
- When the mouse button is released, we will update the size of the bounding rectangle one last time and then call the `handleEvent` method to inform all the registered controllers.

These three steps are performed in the `mousePressed`, `mouseDragged`, and `mouseReleased` methods, respectively, found in lines 121–125, 144–147, and 127–132 of Listing 13-23. All three use the `getPoint` method in the event object to find out where the mouse was when the event occurred.

Of course, the component should provide feedback on which tokens have been selected. This is accomplished in the `paintComponent` method. Lines 71–75 draw the bounding rectangle, and lines 82–84 determines if it surrounds the token currently being drawn. If it does, an instance variable is incremented, and the token's color is changed to yellow. An accessor method, `getNumSelected`, is provided to allow clients to get the number of selected tokens.

Listing 13-23: *An interactive component that allows the user to select a number of tokens*

FIND THE CODE

ch13/nimMultiView/

```
1  import javax.swing.JComponent;
2  import java.awt.Graphics;
3  import java.awt.Insets;
4  import java.awt.Dimension;
5  import java.awt.Point;
6  import java.awt.Color;
7  import java.awt.Rectangle;
8  import java.awt.event.MouseListener;
9  import java.awt.event.MouseMotionListener;
10 import java.awt.event.MouseEvent;
11 import java.awt.event.ActionListener;
12 import java.awt.event.ActionEvent;
13 import java.util.ArrayList;
14
15 /** A component that displays a pile of tokens and allows the user to select a number of
16  *  them. It informs registered listeners when tokens have been selected. Allows the
17  *  client to change the number of tokens in the pile.
18  *
19  *  @author Byron Weber Becker */
20 public class PileComponent2 extends JComponent
21 { // Store the controllers to inform when a selection takes place.
```

Listing 13-23: *An interactive component that allows the user to select a number of tokens* (continued)

```
22    private ArrayList<ActionListener> actionListeners =
23                    new ArrayList<ActionListener>();
24
25    // Information for painting the component.
26    private int numTokens = 0;
27    private int maxTokens;
28
29    private Rectangle selection = null;       // selected area
30    private int numSelected = 0;              // # tokens in selected area
31
32    /** Create a new component.
33    *  @param maxTokens The maximum number of tokens that can be displayed. */
34    public PileComponent2(int maxTokens)
35    { super();
36      this.maxTokens = maxTokens;
37      this.setMinimumSize(new Dimension(40, 60));
38      this.setPreferredSize(new Dimension(60, 90));
39
40      // Add the mouse listener.
41      this.addMouseListener(new MListener());
42      this.addMouseMotionListener(new MMListener());
43    }
44
45    /** Add an action listener to this component's list of listeners. */
46    public void addActionListener(ActionListener listener)
47    { this.actionListeners.add(listener);
48    }
49
50    /** Set the size of the pile.
51    *  @param num The new pile size. 0 <= num <= maxTokens */
52    public void setPileSize(int num)
53    { if (num < 0 || num > this.maxTokens)
54      { throw new IllegalArgumentException("too many/few tokens");
55      }
56      this.numTokens = num;
57      this.selection = null;
58      this.numSelected = 0;
59      this.repaint();
60    }
61
62    /** Paint the component. */
63    public void paintComponent(Graphics g)
```

Listing 13-23: *An interactive component that allows the user to select a number of tokens* (continued)

```
64   { // Values to use in painting.
65     int width = this.getWidth();
66     int height = this.getHeight();
67     int tokenDia = Math.min(width, height/this.maxTokens);
68     int tokenLeft = width/2 - tokenDia;
69
70     // Draw the selection rectangle, if there is one.
71     g.setColor(Color.BLACK);
72     if (this.selection != null)
73     { Rectangle sel = this.selection;
74       g.drawRect(sel.x, sel.y, sel.width, sel.height);
75     }
76
77     // Draw the tokens. Detect which ones are selected. Count them
78     // and color them differently.
79     this.numSelected = 0;
80     for (int i = 0; i < this.numTokens; i++)
81     { int top = height - (i + 1) * tokenDia;
82       if (this.selection != null &&
83           this.selection.contains(tokenLeft + tokenDia / 2,
84                                   top + tokenDia / 2))
85       { this.numSelected++;
86         g.setColor(Color.YELLOW);
87       } else
88       { g.setColor(Color.BLACK);
89       }
90
91       g.fillOval(tokenLeft, top, tokenDia, tokenDia);
92     }
93   }
94
95   /** Get the number of tokens currently selected.
96   *  @return the number of tokens currently selected */
97   public int getNumSelected()
98   { return this.numSelected;
99   }
100
101  /** A helper method to inform all listeners that a selection has been made. */
102  private void handleEvent()
103  { ActionEvent evt = new ActionEvent(
104                      this, ActionEvent.ACTION_PERFORMED, "");
```

Listing 13-23: *An interactive component that allows the user to select a number of tokens* (continued)

```
105     for (ActionListener al : this.actionListeners)
106     { al.actionPerformed(evt);
107     }
108   }
109
110   /** Adjust the selection's size. */
111   private void adjustSelectionSize(Point mPos)
112   { this.selection.setSize(mPos.x - this.selection.x,
113                                    mPos.y - this.selection.y);
114     this.repaint();
115   }
116
117   /** Listen for mouse events within the pile. */
118   private class MListener extends Object
119                                    implements MouseListener
120   {
121     /** A mousePressed event signals the beginning of a selection. */
122     public void mousePressed(MouseEvent e)
123     { PileComponent2.this.selection =
124                                 new Rectangle(e.getPoint());
125     }
126
127     /** A mouseReleased event signals the end of a selection. Finish up the
128      *  selection and inform the listeners. */
129     public void mouseReleased(MouseEvent e)
130     { PileComponent2.this.adjustSelectionSize(e.getPoint());
131       PileComponent2.this.handleEvent();
132     }
133
134     // Required by MouseListener but not needed in this program.
135     public void mouseClicked(MouseEvent e)   {}
136     public void mouseEntered(MouseEvent e)   {}
137     public void mouseExited(MouseEvent e)    {}
138   }
139
140   /**  Listen for mouse events within the pile. */
141   private class MMListener extends Object
142                                 implements MouseMotionListener
143   {
```

Listing 13-23: *An interactive component that allows the user to select a number of tokens* (continued)

```
144       /** The bounds of the selection's rectangle changed. Adjust it. */
145       public void mouseDragged(MouseEvent e)
146       { PileComponent2.this.adjustSelectionSize(e.getPoint());
147       }
148
149       // Required by MouseMotionListener but not needed in this program.
150       public void mouseMoved(MouseEvent e)      {}
151    }
152 }
```

13.9 Patterns

13.9.1 The Model-View-Controller Pattern

Name: Model-View-Controller

Context: A program requires a graphical user interface to interact with the user. You want to program it with the good software engineering principles of encapsulation, information hiding, high cohesion, and low coupling to facilitate future changes.

Solution: Organize the program into a model with one or more views and controllers. The model abstracts the problem the program is designed to solve. Each view displays some part of the model to the user, while controllers translate user actions in a view into method calls on the model.

The Model-View-Controller pattern requires three templates: one for the model, one for the combination of a view and a controller, and one for the main method. Listing 13-13 contains an excellent start on a template for views, but needs an inner class for a controller. Listing 13-1 and Listing 13-3 can be generalized for the model's template and the main method's template, respectively.

LOOKING AHEAD

Written Exercise 13.2 asks you to prepare these templates.

Consequences: Because the model depends only on objects implementing the `IView` interface, coupling is extremely low. The interface can be changed or even completely replaced, usually without changing the model.

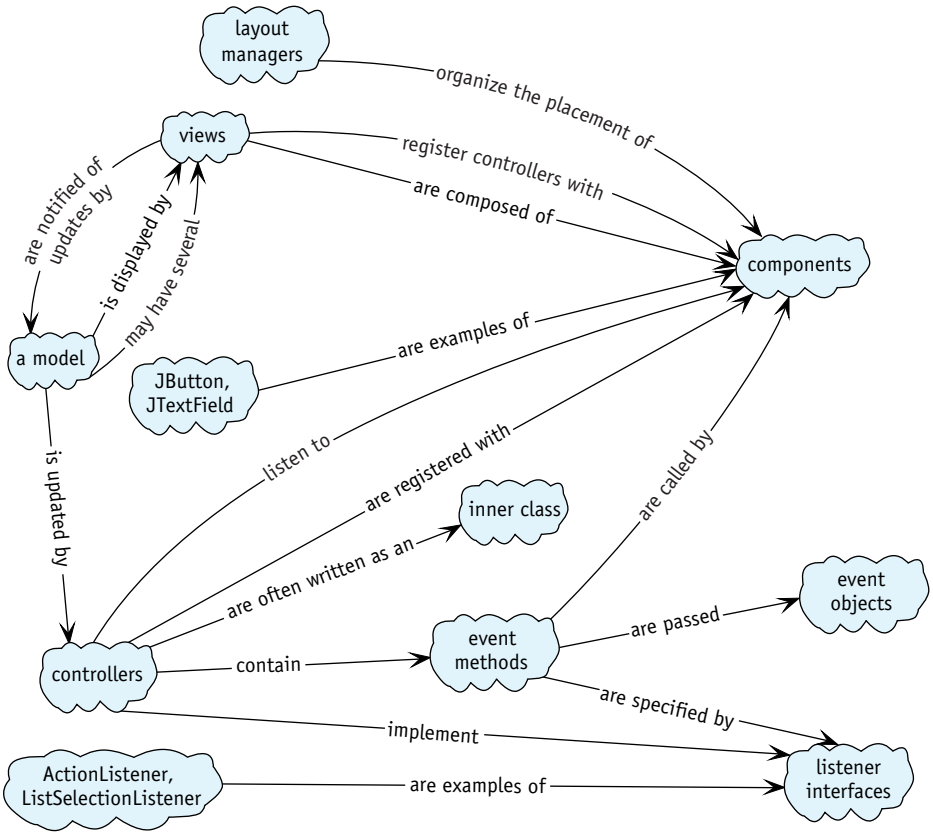
Related Patterns:

- The Extended Class pattern is used by the views when they extend `JPanel`.
- The Has-a (Composition) pattern is used to relate the model to the views and the views to the model.
- The Process All Elements pattern is used to update all of the views with changes in the model.
- The Strategy pattern is used to lay out the view's components and to provide a controller (listener) that reacts appropriately to events in a particular component.

13.10 Summary and Concept Map

Graphical user interfaces use a library of objects, commonly called components, to interact with users. The program is organized into a model containing the abstractions related to the problem, views that display the model to the user, and controllers that interpret user actions to modify the model. One model may have several views, and each view may have several controllers.

It is also possible to create components to perform specific tasks for which no existing component is available.



13.11 Problem Set

Written Exercises

13.1 Explain how using subviews (Section 13.5) is good software engineering. Refer specifically to the concepts of cohesion and coupling.

13.2 Write the three code templates required for the Model-View-Controller pattern. Listing 13-13 contains an excellent start on a template for views, but needs an inner class for a controller. Listing 13-1 and Listing 13-3 can be generalized for the model's template and the main method's template, respectively.

13.3 Prepare a class diagram showing the relationships between the classes in the Model-View-Controller pattern. Assume the controller has been written in a separate class, as shown in Listing 13-9, and implements an `ActionListener`.

13.4 List the signatures for all the methods required to implement a `WindowListener`.

13.5 The Java library contains two classes named `MouseAdapter` and `MouseMotionAdapter`. Discuss how they could be used to simplify the `PileComponent2` class shown in Listing 13-23.

13.6 The Java library contains an interface named `MouseInputListener`. Examine the documentation and discuss how it could be used in the `PileComponent2` class shown in Listing 13-23.

Programming Exercises

13.7 Find the code for the version of Nim with multiple views.

a. Add a new view whose function is to offer hints to the current player. (*Hint*: Assuming the rules where 1, 2, or 3 tokens may be removed, a player who leaves 1, 2, or 3 tokens for his or her opponent has made a serious mistake. Similarly, a player who leaves exactly four tokens is in a very strong position. Generalize these observations.)

b. Modify the `NimPlayerView` class to use a `JComboBox` for user input instead of `JButton` objects.

c. Modify the `NimPlayerView` class to use a `JSlider` for user input.

d. Add a new view whose function is to start a new game. The user should be able to specify who starts and how large the initial pile of tokens should be. The player should also be able to start a new game with the program choosing either or both of these values randomly.

e. Views do not actually need to belong to a graphical user interface. Write a class named `NimLogger` that implements `IView`. Modify the `Nim` program to use `NimLogger` to write the state of the game after each move to a file. (*Hint*: You should *not* extend `JPanel` or include any classes from the `javax.swing` or `java.awt` packages. Create the `NimLogger` object in the `main` method.)

f. Modify the model and the view so users may remove up to half of the remaining tokens in each turn. Start the game with a random pile of 20 to 30 tokens. The existing views with three buttons each are inappropriate. Design a new view.

g. Modify the `PileComponent2` class to show the tokens as a block, three tokens wide. The top row of the block may have less than three tokens.

h. The `PileComponent2` class shown in Listing 13-23 does not work when a user clicks and drags the mouse upward or leftward. The problem is that the width or the length of the selection rectangle becomes negative, resulting in an "empty" rectangle. Fix this problem.

i. The `PileComponent2` class shown in Listing 13-23 currently allows the user to select any number of visible tokens, even though the game only allows a maximum of three tokens to be removed. Fix the component so that the selection rectangle is not allowed to enclose more than three tokens.

13.8 Find the code displayed in Listing 13-21. Write a simple `main` method to display it in a frame. Observe that it is possible to select several items at once using the Shift or Control keys.

a. Modify the program to print all of the items that have been selected.

b. Modify the program so users can select only one item at a time.

c. The `JList` documentation includes sample code for a class named `MyCellRenderer`. Read the documentation, and then change the program so that each element of the list is displayed using the appropriate color.

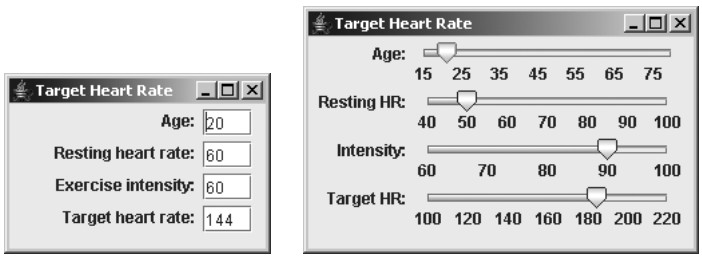
Programming Projects

13.9 Write a program to assist users in calculating their target heart rate for an exercise program. You can find many formulas on the Web for calculating target heart rate. One is based on the user's age, resting heart rate, and targeted intensity: *intensity* * (220 – *age* – *restingHR*) + *restingHR*, where *intensity* is a percentage (typically 80 to 90%), *age* is the user's age in years, and *restingHR* is the user's resting heart rate in beats per minute. The model will have mutator methods for *intensity*, *age*, and *restingHR*, and accessor methods for those three plus the target heart rate.

Two possible views are shown in Figure 13-17.

a. Write the program's view using `JTextField` components.

b. Write the program's view using `JSlider` components.



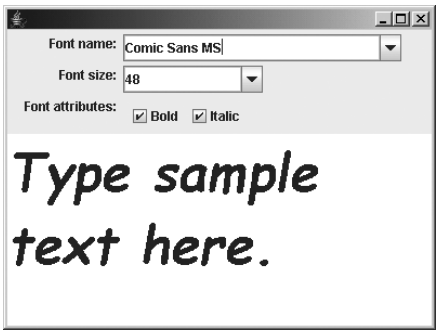
(figure 13-17)

Two possible views for a target heart rate calculator

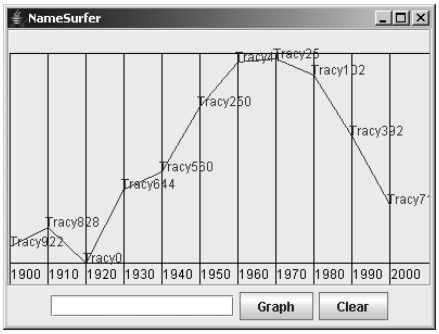
13.10 Write a program that allows you to display font samples. A proposed user interface is shown in Figure 13-18. The model for this program will have methods such as `setFontName`, `setFontSize`, `setBold`, `setItalic`, and `getFont`. The components used in the interface include `JComboBox`, `JCheckBox`, `JTextArea`, and `becker.gui.FormLayout`.

(figure 13-18)

Two interfaces



An interface for generating font samples



An interface for plotting the popularity of names through time

13.11 Explore the documentation for the `becker.xtras.nameSurfer` package.[5] In particular, see the package overview and Figure 13-18 for an example of the interface.

a. Write a model named `SurferModel`. Demonstrate your model, working with classes from the `nameSurfer` package to form a complete program.

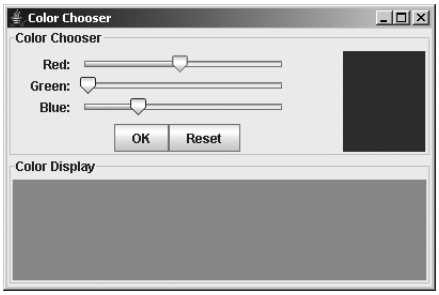
b. Write a view named `SurferView`. Demonstrate your view, working with classes from the `nameSurfer` package to form a complete program. (*Hint*: You will need to implement a custom component to draw the graph.)

13.12 Implement a view to choose a color. Figure 13-19 has three `JSlider` components, one for each of the red, green, and blue parts of a color. Their values range between 0 and 255. Use an empty `JPanel` to display the current color as the sliders are moved by calling the panel's `setBackground` method.

Demonstrate your view with a simple program. The model will have two methods: `setColor` and `getColor`. `setColor` is called when the OK button is pressed, resulting in a second view being updated with the chosen color.

(figure 13-19)

Sample interfaces for a color chooser and a Web browser



Simple color chooser

Simple Web browser

[5] The original idea for this problem is attributed to Nick Parlante at Stanford University.

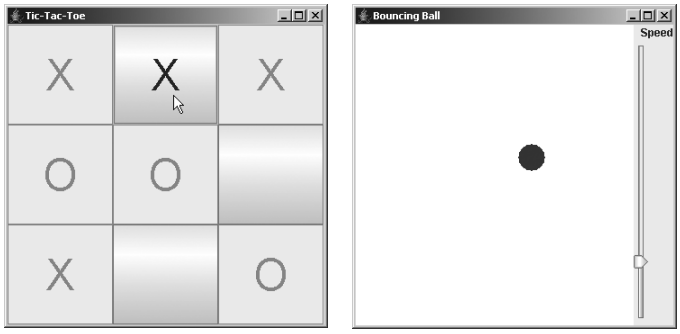
13.13 Use the `JEditorPane` to implement a simple Web browser like the one shown in Figure 13-19. Users should be able to type a URL into a text field and have it displayed in the `JEditorPane`. Your browser should also correctly follow links to display a new page. The `JEditorPane` may not be editable for links to work. The model for the browser will be the current URL to display. Enhancements may require adding a history list and other features to the model.

a. Add scroll bars to the `JEditorPane` that show only if needed.

b. Add a toolbar with Forward, Back, and Home buttons.

c. Use a `JComboBox` for entering URLs. Add URLs the user has typed to the `JComboBox` for easier selection in the future.

13.14 Implement the game of Tic-Tac-Toe for two users (see Figure 13-20). Search the Web for the rules if you are unfamiliar with the game. Use a button for each of the nine squares to gather input from the users. Disable the buttons and change their labels as they are played. When the mouse is moved over an unplayed square, show either X or O, depending on whose turn it is. Announce the winner with a dialog box and start a new game.



Tic-tac-toe game

Animated bouncing ball

(figure 13-20)

Sample interfaces for a game and an animation

13.15 Write a program that displays a bouncing ball and allows for its speed to be changed and the size of the box it bounces in to be changed (see Figure 13-20). Note the following hints:

- Read the documentation for the `javax.swing.Timer` class. An appropriate delay is 1000/30. There are several classes named `Timer`; be sure to read the right one.
- Write a `BallModel` class with methods such as `getBallBounds` and `setBoxBounds`. The `java.awt.Rectangle` class is convenient for maintaining size and position information for both the ball and the box. The `BallModel` will also contain an instance of `Timer`, updating the position of the ball every time it "ticks."

- The `BallView` class should contain a custom component to draw the ball. It will need a controller implementing `ComponentListener` to resize the model's box when the component is resized.
- The `BallView` class should also contain an instance of `JSlider` to adjust the speed of the bouncing ball.

13.16 Implement a model for a right triangle. It will have two methods to set the base and the height but will calculate the length of the hypotenuse using the Pythagorean theorem ($a^2 + b^2 = c^2$). It will also have three methods to get the length of each side. The length of the base and the height must be between 1 and 100, inclusive. Figure 13-21 shows several different views of the model.

a. Implement a view using `JTextField` components.

b. Implement a view using `JSlider` components.

c. Implement a view using a `JButton` to increment the length of the base and another to decrement it. Do so similarly for the height.

d. Implement a view using `JSpinner` to adjust the base and height.

e. Implement a view using `JCombobox` or `JList` that allows the user to select one of several standard triangle sizes.

f. Implement a custom component that draws a picture of the triangle. Set the size of the triangle using one of the other views.

g. Implement a custom component that draws a picture of the triangle. Add a controller for the mouse that detects clicks on the triangle. When the triangle is clicked, paint "handles" to show that it is selected. Allow the user to change its size by dragging the handles.

h. Implement a view showing several of the preceding views other than (e). Be sure that they all display the same information about the triangle model.

(figure 13-21)

Several views of a triangle model