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

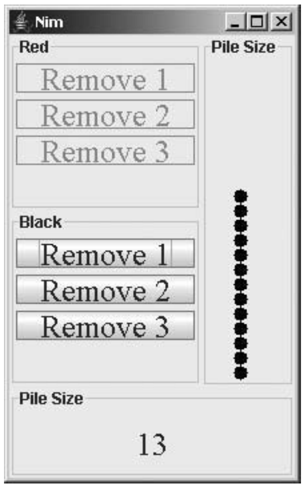
Building the graphical user interface (GUI) for a program can be one of the more rewarding parts of programming. Finally, we begin to *see* the results of our labor and are able to manipulate our program directly. The user interface is also a place where we can use aesthetic skills and sensibilities.

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)

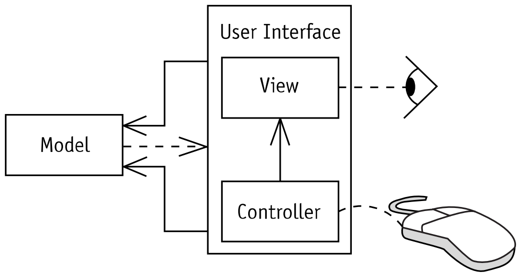
Requirements for the game of Nim

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.

PATTERN

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

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.

Build the View and Controllers

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

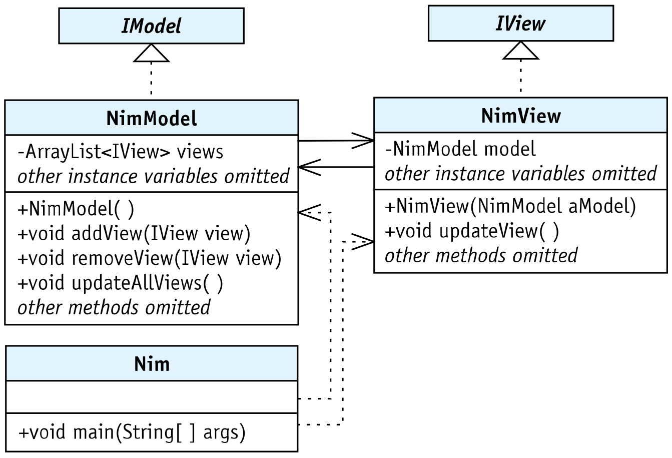
We will elaborate on these steps in each of the next three subsections.

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`


IModel
IView
NimModel
-ArrayList<IView> views
other instance variables omitted
+NimModel( )
+void addView(IView view)
+void removeView(IView view)
+void updateAllViews( )
other methods omitted
NimView
-NimModel model
other instance variables omitted
+NimView(NimModel aModel)
+void updateView( )
other methods omitted
Nim
+void main(String[ ] args)

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*

```
import becker.util.IModel;
import becker.util.IView;
import java.util.ArrayList;

/** A class implementing a version of Nim. There is a (virtual) pile of tokens. Two
 *  players take turns removing 1, 2, or 3 tokens. The player who takes the last token
 *  wins the game.
 *
 *  @author Byron Weber Becker */
public class NimModel extends Object implements IModel
```

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

```
{ private ArrayList<IView> views = new ArrayList<IView>();

  /** Construct a new instance of the game of Nim. */
  public NimModel()
  { super();
  }

  /** Add a view to display information about this model.
   *  @param view The view to add. */
  public void addView(IView view)
  { this.views.add(view);
  }

  /** Remove a view that has been displaying information about this model.
   *  @param view The view to remove. */
  public void removeView(IView view)
  { this.views.remove(view);
  }

  /** Inform all the views currently displaying information about this model that the
   *  model has changed and their display may need changing too. */
  public void updateAllViews()
  { for (IView view : this.views)
    { view.updateView();
    }
  }
}
```

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.

FIND THE CODE

ch13/nim Infrastructure/

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 }
```

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*

```
import javax.swing.JFrame;

/** Run the game of Nim. There is a (virtual) pile of tokens. Two players take turns
 *   removing 1, 2, or 3 tokens. The player who takes the last token wins the game.
 *
 *   @author Byron Weber Becker */
public class Nim
{
  public static void main(String[ ] args)
  { NimModel model = new NimModel();
    NimView view = new NimView(model);

    JFrame f = new JFrame("Nim");
    f.setSize(250, 200);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setContentPane(view);
    f.setVisible(true);
  }
}
```

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*

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

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)

```
   /** We need a way to create a nonrandom game for testing purposes. */
   private NimModel(int pileSize, Player next)
   { super();
     this.pileSize = pileSize;
     this.whoseTurn = next;
   }

   /** Generate a random number between two bounds. */
   private static int random(int lower, int upper)
   { return (int)(Math.random()*(upper-lower+1)) + lower;
   }

   /** Choose a player at random.
   *  @return Player.RED or Player.BLACK with 50% probability for each */
   private static Player chooseRandomPlayer()
   { if (Math.random() < 0.5)
     { return Player.RED;
     } else
     { return Player.BLACK;
     }
   }

   /** Get the current size of the pile.
   *  @return the current size of the pile */
   public int getPileSize()
   { return this.pileSize;
   }

   /** Get the next player to move.
   *  @return Either Player.RED or Player.BLACK if the game has not yet been won,
   *   or Player.NOBODY if the game has been won. */
   public Player getWhoseTurn()
   { return this.whoseTurn;
   }

   /** Get the winner of the game.
   *  @return Either Player.RED or Player.BLACK if the game has already been won;
   *  Player.NOBODY if the game is still in progress. */
   public Player getWinner()
   { return this.winner;
   }
```

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

```

/** Is the game over?
 *  @return true if the game is over; false otherwise. */
private boolean gameOver()
{ return this.pileSize == 0;
}

/** Remove one, two, or three tokens from the pile. Ignore any attempts to take
 *  too many or too few tokens. Otherwise, remove howMany tokens from the pile
 *  and update whose turn is next.
 *  @param howMany How many tokens to remove.
 *  @throws IllegalStateException if the game has already been won */
public void removeTokens(int howMany)
{ if (this.gameOver())
  { throw new IllegalStateException(
                                "The game has already been won.");
  }

  if (this.isLegalMove(howMany))
  { this.pileSize = this.pileSize - howMany;
    if (this.gameOver())
    { this.winner = this.whoseTurn;
      this.whoseTurn = Player.NOBODY;
    } else
    { this.whoseTurn =
                NimModel.otherPlayer(this.whoseTurn);
    }
    this.updateAllViews();
  }
}

// Is howMany a legal number of tokens to take?
private boolean isLegalMove(int howMany)
{ return howMany >= 1 && howMany <= MAX_REMOVE &&
         howMany <= this.pileSize;
}

// Return the other player.
private static Player otherPlayer(Player who)
{ if (who == Player.RED)
  { return Player.BLACK;
  } else if (who == Player.BLACK)
  { return Player.RED;
```

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

```
    } else
    { throw new IllegalArgumentException();
    }
  }

  // The addView, removeView, and updateAllViews methods could be included
  // here. That isn't necessary in this case because NimModel extends AbstractModel.

  /** Test the class. */
  public static void main(String[] args)
  { System.out.println("Testing NimModel");
    NimModel nim = new NimModel(10, Player.RED);
    Test.ckEquals("pile size", 10, nim.getPileSize());
    Test.ckEquals("winner", Player.NOBODY, nim.getWinner());
    Test.ckEquals("next", Player.RED, nim.getWhoseTurn());

    /** ------ find the code to see complete test suite ------*/
  }
}
```

13.4 Building the View and Controllers

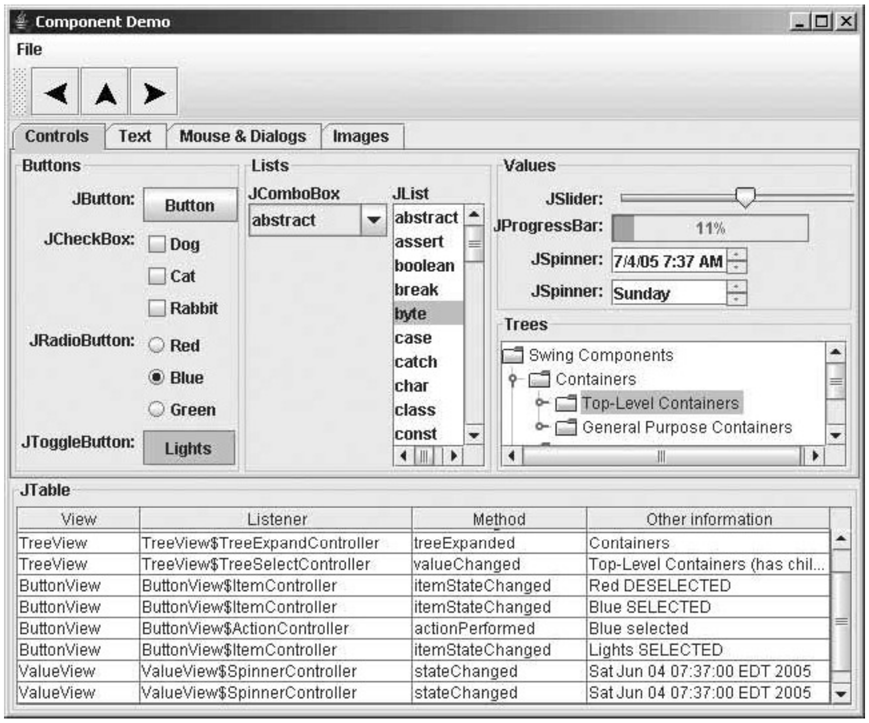
The view, of course, is what displays information from the model to the user. It is the visible part of the user interface. The controllers are what make the interface interactive. They listen for the user manipulating controls such as buttons or menus and then make appropriate calls to the commands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



JComboBox: abstract

JProgressBar: 11%

JSpinner: 7/4/05 7:37 AM

JSpinner: Sunday

View	Listener	Method	Other information
TreeView	TreeView$TreeExpandController	treeExpanded	Containers
TreeView	TreeView$TreeSelectController	valueChanged	Top-Level Containers (has chil...
ButtonView	ButtonView$ItemController	itemStateChanged	Red DESELECTED
ButtonView	ButtonView$ItemController	itemStateChanged	Blue SELECTED
ButtonView	ButtonView$ActionController	actionPerformed	Blue selected
ButtonView	ButtonView$ItemController	itemStateChanged	Lights SELECTED
ValueView	ValueView$SpinnerController	stateChanged	Sat Jun 04 07:37:00 EDT 2005
ValueView	ValueView$SpinnerController	stateChanged	Sat Jun 04 07:37:00 EDT 2005

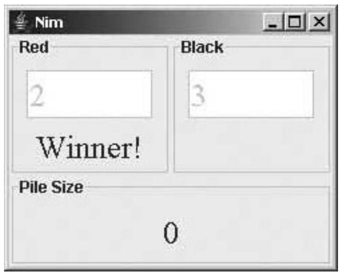
(figure 13-5)

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.



Red: 2 — Winner!

Black: 3

Pile Size: 0

(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 `JTextField`s to accept input from the players
- One `JLabel` showing the pile's current size
- Two `JLabel`s 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

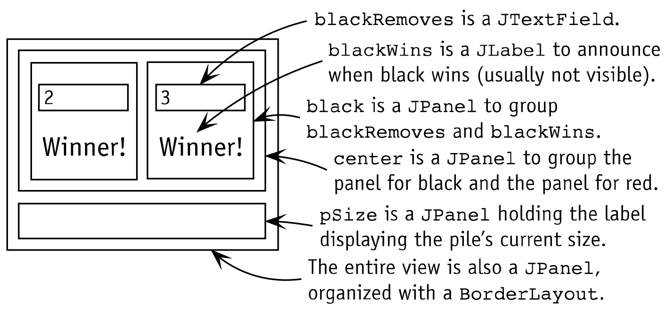
Layout managers were discussed in Section 12.6.

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 `JPanel`s, as shown in Figure 13-7.

(figure 13-7)

`NimView` *uses nested* `JPanel`s *to lay out the components*


2
3
Winner!
Winner!
blackRemoves is a JTextField.
blackWins is a JLabel to announce when black wins (usually not visible).
black is a JPanel to group blackRemoves and blackWins.
center is a JPanel to group the panel for black and the panel for red.
pSize is a JPanel holding the label displaying the pile's current size.
The entire view is also a JPanel, organized with a BorderLayout.

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 JPanels` 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.

FIND THE CODE

ch13/nimOneView/

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);
```

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

```
      red.setBorder(BorderFactory.createTitledBorder("Red"));

      // A panel for the black player.
      JPanel black = new JPanel();
      black.add(this.blackRemoves);
      black.add(this.blackWins);
      black.setBorder(BorderFactory.createTitledBorder("Black"));

      // Pile size information.
      JPanel pSize = new JPanel();
      pSize.add(this.pileSize);
      pSize.setBorder(
            BorderFactory.createTitledBorder("Pile Size"));

      // Group the red and black panels.
      JPanel center = new JPanel();
      center.setLayout(new GridLayout(1, 2));
      center.add(red);
      center.add(black);

      // Lay out the pieces in this view.
      this.setLayout(new BorderLayout());
      this.add(center, BorderLayout.CENTER);
      this.add(pSize, BorderLayout.SOUTH);
   }
}
```

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*

```
public class NimView extends JPanel implements IView
{ private NimModel model;
  private JTextField redRemoves = new JTextField(5);
  // Other instance variables, constructor, and methods omitted.

  /** Called by the model when it changes. Update the information this view displays. */
  public void updateView()
  { // Update the size of the pile.
    this.pileSize.setText("" + this.model.getPileSize());

    // Enable and disable the text fields for each player.
    this.redRemoves.setEnabled(
            this.model.getWhoseTurn() == Player.RED);
    this.blackRemoves.setEnabled(
            this.model.getWhoseTurn() == Player.BLACK);

    // Proclaim the winner, if there is one.
    this.redWins.setVisible(
            this.model.getWinner() == Player.RED);
    this.blackWins.setVisible(
            this.model.getWinner() == Player.BLACK);
  }
}
```

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`

```
public class JTextField extends ...
{ private ActionListener actionListener;
  private FocusListener focusListener;

  public void addActionListener(ActionListener aListener)
  { this.actionListener = aListener;
  }

  public void addFocusListener(FocusListener fListener)
  { this.focusListener = fListener;
  }

  private void handleEvent()
  { 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);
}
```

LOOKING AHEAD

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

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.

FIND THE CODE

ch13/nimOneView/

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);
```

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

```
      } catch (NumberFormatException ex)
      { this.textfield.selectAll();
      }
    }
  }
```

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 `addXxxListener` 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*

```
public class NimView extends JPanel implements IView
{ // Instance variables omitted.

  public NimView()
  { // Some details omitted.
    this.registerControllers();
  }

  /** Register controllers for the components the user can manipulate. */
  private void registerControllers()
  { RemoveController redController =
        new RemoveController(this.model, this.redRemoves);
    this.redRemoves.addActionListener(redController);
```

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

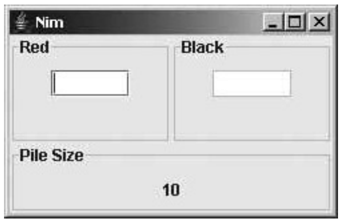
```

      this.blackRemoves.addActionListener(
         new RemoveController(this.model, this.blackRemoves));
   }
}
```

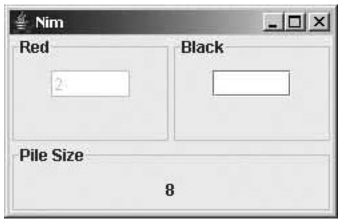
If you run the program with these additions, you should be able to play a complete, legal game, as shown in Figure 13-8.

(figure 13-8)

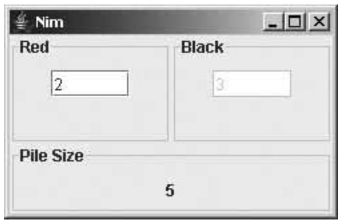
User interface as it appears at each stage of a complete game



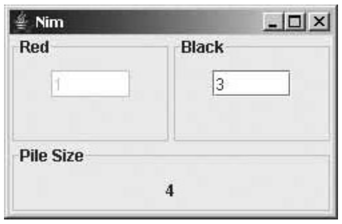
a) The game begins with a pile of 10. Red has the first turn.



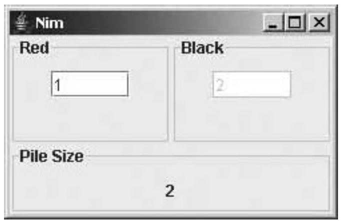
b) Red takes two tokens; now it's black's turn. The player must click in its text field before entering a value.



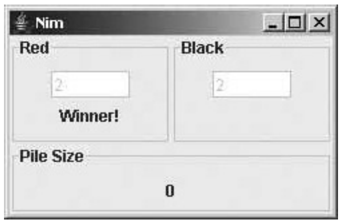
c) Black takes three tokens. It's red's turn. The "2" from red's previous turn still shows. Red does *not* need to click in its text field before entering a value but must delete the old value before entering a new one.



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.

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 `JLabels` 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.

FIND THE CODE

ch13/fonts/

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

```
import java.awt.Font;
import java.awt.GraphicsEnvironment;

/** List the font names available on the current computer system.
 *
 *  @author Byron Weber Becker */
public class ListFonts extends Object
{ public static void main(String[] args)
  { GraphicsEnvironment ge =
        GraphicsEnvironment.getLocalGraphicsEnvironment();
    Font[] names = ge.getAllFonts();

    for (Font f : names)
    { System.out.println(f.getName());
    }
  }
}
```

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*

```
import javax.swing.JPanel;
import becker.util.IView;
import javax.swing.JTextField;
import javax.swing.JLabel;
import javax.swing.BorderFactory;
import java.awt.GridLayout;
import java.awt.BorderLayout;
import java.awt.Font;

/** Provide a view of the game of Nim to a user.
 *
 *  @author Byron Weber Becker */
public class NimView extends JPanel implements IView
{ // The model implementing Nim's logic.
  private NimModel model;

  // 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!");

  /** Construct the view.
   *  @param aModel The model we will be displaying. */
  public NimView(NimModel aModel)
  { super();
    this.model = aModel;

    this.layoutView();
    this.registerControllers();

```

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

```
      this.model.addView(this);
      this.updateView();
   }

   /** Called by the model when it changes. Update the information this view displays. */
   public void updateView()
   { this.pileSize.setText("" + this.model.getPileSize());

      this.redRemoves.setEnabled(
            this.model.getWhoseTurn() == Player.RED);
      this.blackRemoves.setEnabled(
            this.model.getWhoseTurn() == Player.BLACK);
      this.redWins.setVisible(
            this.model.getWinner() == Player.RED);
      this.blackWins.setVisible(
            this.model.getWinner() == Player.BLACK);

      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("");
      }
   }

   /** Layout the view. */
   private void layoutView()
   { // A panel for the red player
      JPanel red = new JPanel();
      red.add(this.redRemoves);
      red.add(this.redWins);
      red.setBorder(BorderFactory.createTitledBorder("Red"));

      // A panel for the black player
      JPanel black = new JPanel();
      black.add(this.blackRemoves);
      black.add(this.blackWins);
      black.setBorder(BorderFactory.createTitledBorder("Black"));

      // Pilesize info.
      JPanel pSize = new JPanel();
```

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

```
    pSize.add(this.pileSize);
    pSize.setBorder(
          BorderFactory.createTitledBorder("Pile Size"));

    // Group the red and black panels.
    JPanel center = new JPanel();
    center.setLayout(new GridLayout(1, 2));
    center.add(red);
    center.add(black);

    // Lay out the pieces in this view.
    this.setLayout(new BorderLayout());
    this.add(center, BorderLayout.CENTER);
    this.add(pSize, BorderLayout.SOUTH);

    // 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);
  }

  /** Register controllers for the components the user can manipulate. */
  private void registerControllers()
  { this.redRemoves.addActionListener(
        new RemovesController(this.model, this.redRemoves));
    this.blackRemoves.addActionListener(
      new RemovesController(this.model, this.blackRemoves));
  }
}
```

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*

```
import becker.util.IView;
import javax.swing.JPanel;
«list of other imports»

public class «viewName» extends JPanel implements IView
{ private «modelClassName» model;

  «component declarations»

  public «viewName»(«modelClassName» aModel)
  { super();
    this.model = aModel;
    this.layoutView();
    this.registerControllers();
    this.model.addView(this);
    this.updateView();
  }

  public void updateView()
  { «statements to update the components in the view»
  }

  private void layoutView()
  { «statements to lay out the components within the view»
  }

  private void registerControllers()
  { «statements to construct and register controllers»
  }
}
```

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,