FIND THE CODE

ch12/inventory/

Listing 12-19: *A class using interfaces to promote flexibility*

```
 1 public class Inventory extends Object
 2 { private List<Item> inventory = new ArrayList<Item>();
 3   private List<Item> reorder = new LinkedList<Item>();
 4   ...
 5
 6   /**Remove the specified items from the current inventory. Update the list of items
 7    *  to reorder.
 8    *  @itemsSold  The items that have been sold and need to be removed from inventory. */
 9   public void removeInventory(List<Item> itemsSold)
10   { for (Item item : itemsSold)
11     {
12       // Remove the item from the inventory.
13       this.inventory.remove(item);
14
15       // If it's the last one and not already on the reorder list, add it
16       if (!this.inventory.contains(item) &&
17                 !this.reorder.contains(item))
18       { this.reorder.add(item);
19       }
20     }
21   }
22 }
```

By using `List` to declare variables in lines 2, 3, and 9, the programmer has left lots of flexibility to change the actual classes being used. For example, the `ArrayList` in line 2 could be changed to a `LinkedList` with no further changes in the rest of the program.

12.6 GUI: Layout Managers

PATTERN

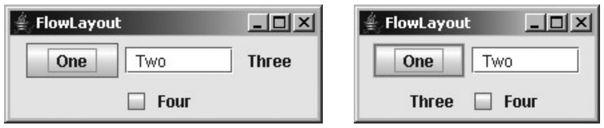
Strategy

Most graphical user interfaces allow users to interact with many components (buttons, text boxes, sliders, and so on). The issue to be addressed in this section is how Java arranges the components in a panel, both initially and as the user resizes the frame displaying the panel. The task of arranging the components on the panel is called **layout**. Java uses strategy objects called **layout managers** to determine how to arrange the components. By using strategy objects, `JPanel` can display the same set of components in many different arrangements.

12.6.1 The `FlowLayout` Strategy

The default layout strategy for a `JPanel` is an instance of `FlowLayout`. It adds components to the current row until there is no more room. It then starts a new row. The length of a row is determined by the width of the `JPanel`. Wider panels will have more components on a row.

The left image in Figure 12-18 shows four components organized with a `FlowLayout` strategy. The components are displayed left to right, top to bottom, in the same order they were added. The right image shows how those same components are reorganized when the frame is narrower.



(figure 12-18)

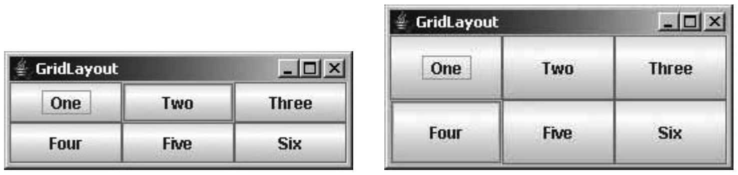
The `FlowLayout` *strategy*

A `FlowLayout` object centers rows by default. It can also be set to align them on either the left or right side of the panel.

Each component has a preferred size, which is respected by `FlowLayout`. As we'll soon see, some layout managers ignore such size information.

12.6.2 The `GridLayout` Strategy

The strategy implemented by a `GridLayout` object is to place all of the components into a grid, as shown in Figure 12-19. Each component is made the same size as all the others, completely ignoring their preferred sizes. The number of rows and columns is set when the strategy object is created.



(figure 12-19)

The `GridLayout` *strategy*

Setting a `JPanel`'s layout strategy is done with its `setLayout` method, as shown in lines 17–18 of Listing 12-20. This listing is already showing the program structure we will adopt for our graphical user interfaces. A group of components is combined by extending `JPanel`. Laying out the components is a distinct task that is delegated to a private helper method called `layoutView`.

Listing 12-21 displays an instance of this panel in a frame.

Listing 12-20: *A* `JPanel` *extended to show a group of buttons, organized with a grid strategy*

FIND THE CODE

ch12/layoutManagers/

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

public class DemoGridLayout extends JPanel
{
  private JButton one = new JButton("One");
  private JButton two = new JButton("Two");
  // Instance variables for the last four buttons are omitted.

  public DemoGridLayout()
  { super();
    this.layoutView();
  }

  private void layoutView()
  { // Set the layout strategy to a grid with 2 rows and 3 columns.
    GridLayout strategy = new GridLayout(2, 3);
    this.setLayout(strategy);

    // Add the components.
    this.add(this.one);
    this.add(this.two);
    // Code to add the last four buttons is omitted.
  }
}
```

PATTERN

Strategy

Listing 12-21: *A* `main` *method that displays a custom* `JPanel` *in a frame*

FIND THE CODE

ch12/layoutManagers/

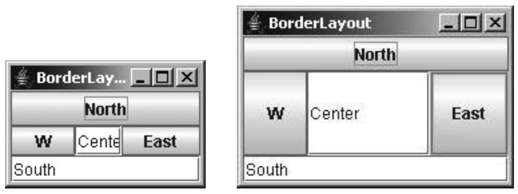
```
import javax.swing.*;

public class GridLayoutMain
{
  public static void main(String[] args)
  { JPanel p = new DemoGridLayout();

    JFrame f = new JFrame("GridLayout");
    f.setContentPane(p);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.pack();               // Base frame size on preferred size of components.
    f.setVisible(true);
  }
}
```

12.6.3 The `BorderLayout` Strategy

The `BorderLayout` strategy lays out up to five objects in a panel, as shown in Figure 12-20. No matter what size the panel is, the north and south areas cover the entire width. Their heights are determined by the preferred heights of the components they hold. The east and west areas expand or contract to occupy the remaining height of the panel. Their widths are determined by the preferred sizes of the components they hold. Finally, the center area expands or contracts to occupy the remaining space.

(figure 12-20)

The `BorderLayout` *strategy*



Areas that do not have a component will not take any space. For example, if the button was left out of the east area in Figure 12-20, the center area would simply expand to fill it.

The layout managers we've seen previously arrange the components according to the order in which they are added to the panel. `BorderLayout` handles positioning with a **constraint**, which is specified when the component is added. The constraint says where the component should be placed.

Listing 12-20 could be modified to use a `BorderLayout` strategy by changing line 17 to:

```
17      BorderLayout strategy = new BorderLayout();
```

and changing the lines that add the components to use the required constraints.

```
21      this.add(this.one, BorderLayout.EAST);
22      this.add(this.two, BorderLayout.NORTH);
```

12.6.4 Other Layout Strategies

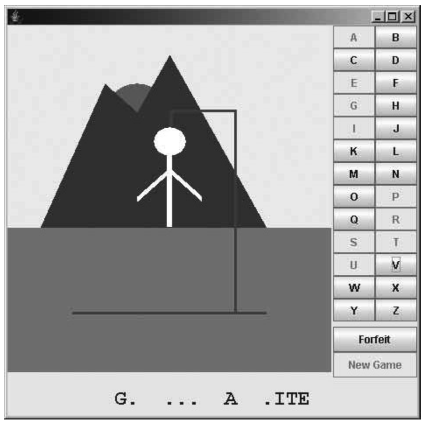
The `BoxLayout` strategy arranges components in a horizontal row or a vertical column. It tries to respect the preferred sizes of components. However, if a component does not have a maximum size, it will grow or shrink to fill available space. Text fields and text areas, for example, do not have a maximum size unless you set one.

Like `GridLayout`, `GridBagLayout` uses a grid. However, its cells can vary in size, and a component can take up more than one cell in the grid. To accomplish all this, it uses a fairly complex constraint, called `GridBagConstraints`.

Another constraint-based layout strategy is `SpringLayout`. It works by specifying how the edges of each component relate to other components or to the edges of the enclosing panel.

12.6.5 Nesting Layout Strategies

A single layout strategy is usually not enough for a complex graphical user interface. Consider Figure 12-21, for example. None of the simpler layout strategies we've covered can handle this by themselves. GridBagLayout and SpringLayout could do it, but using them would involve a tremendous amount of work in setting all the constraints.

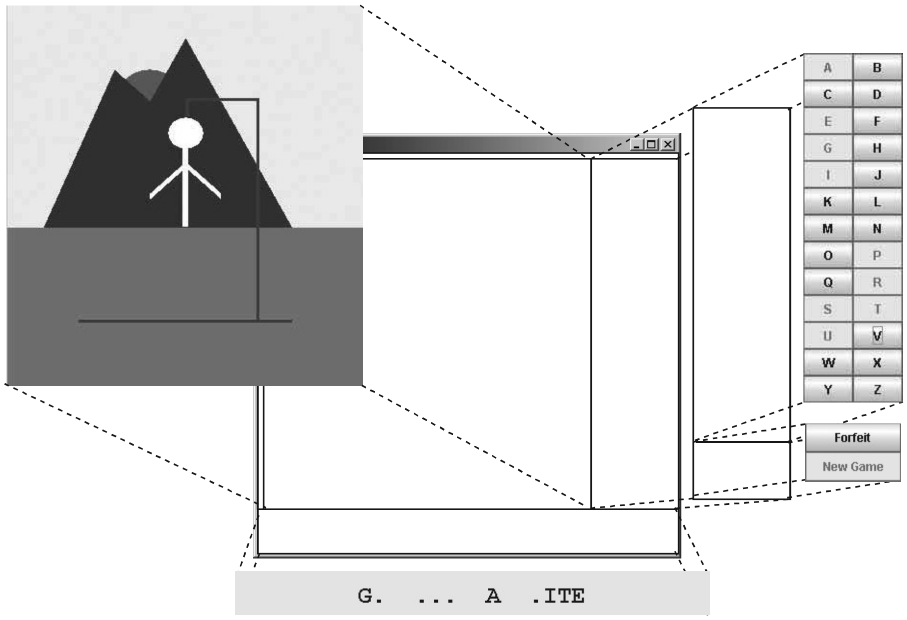


(figure 12-21)
A complex layout task

An excellent solution is based on the fact that JPanel is also a component. It can be added to another JPanel that is organized by its own layout strategy object. The user interface in Figure 12-21 is organized with four JPanel objects, as shown in Figure 12-22.

LOOKING AHEAD

Programming Exercise 12.12 asks you to finish implementing HangmanView.



(figure 12-22)
Laying out a complex user interface using nested panels, each with its own layout strategy

The four `JPanel` objects are as follows:

- `controls` is organized by a `GridLayout` and holds the `Forfeit` and `New Game` buttons.
- `letters` is also organized by a `GridLayout` and holds 26 buttons, one for each letter of the alphabet.
- `buttons` is organized by a `BoxLayout` and holds two `JPanel` components, `controls` and `letters`.
- `hangman` is organized by a `BorderLayout`. The center area holds the graphic showing the gallows. The south area holds a `JLabel` displaying the letters guessed so far. The east area holds the `buttons` panel (which holds `letters` and `controls`). The north and west area of the `BorderLayout` are empty and shrink to take no space.

This interface can be implemented with code similar to that shown in Listing 12-22.

FIND THE CODE

ch12/hangman/

PATTERN

Strategy

Listing 12-22: *Implementing nesting layout managers*

```
import becker.xtras.hangman.*;
import javax.swing.*;
import java.awt.*;

/** Layout the view for the game of hangman.
*
*  @author Byron Weber Becker */
public class HangmanView extends JPanel
{  // Constructor omitted.

   /** Layout the view in a JPanel managed by BorderLayout. */
   private void layoutView()
   { JPanel hangman = this;          // Use same name as previous discussion
     hangman.setLayout(new BorderLayout());

     // South
     JLabel phrase = new JLabel("GO FLY A KITE");
     hangman.add(phrase, BorderLayout.SOUTH);

     // Center
     JComponent gallows = new GallowsView(
                                new SampleHangman());
     hangman.add(gallows, BorderLayout.CENTER);

     // East -- letters and controls
     JPanel buttons = this.buttonsPanel();
```

Listing 12-22: *Implementing nesting layout managers* (continued)

```
      hangman.add(buttons, BorderLayout.EAST);
   }

   /** Layout and return a subpanel with all the buttons. */
   private JPanel buttonsPanel()
   { // A JPanel holding 26 buttons, one for each letter of the alphabet.
      JPanel letters = new JPanel();
      letters.setLayout(new GridLayout(13, 2));
      for (char ch = 'A'; ch <= 'Z'; ch++)
      { letters.add(new JButton("" + ch));
      }

      // A JPanel holding the Forfeit and New Game buttons is omitted.

      return letters;
   }
}
```

12.7 Patterns

12.7.1 The Polymorphic Call Pattern

Name: Polymorphic Call

Context: You are writing a program that handles several variations of the same general idea (for example, several kinds of bank accounts). Each kind of thing has similar behaviors, but the details may differ.

Solution: Use a polymorphic method call so that the actual object being used determines which method is called. The most basic form of the pattern is identical to the Command Invocation pattern from Chapter 1 except for how the *«objReference»* is given its value. For example,

```
«varTypeName» «objReference» = «instance of objTypeName»;
...
«objReference».«serviceName»(«parameterList»);
```

where *«objTypeName»* is a subclass of *«varTypeName»* or *«objTypeName»* is a class that implements the interface *«varTypeName»*.

There are many variations. For example, *«objReference»* could be a simple instance variable, an array, a parameter, or a value returned from a method.