

Widgets

In 1981 Xerox introduced the Star¹. This computer boasted a graphical user interface where users could interact with their information directly on the screen using a mouse and a keyboard. Though many parts of these ideas had occurred earlier, the Star is heralded as the trendsetter for the workstations we know today. In 1983 Apple introduced the Lisa with similar capabilities. The Lisa was costly and slow. However, during the 1984 NFL Super bowl, Apple introduced the Macintosh, which forever changed the way people interact with their computers. Graphical user interfaces were now affordable for everyone.

A unique aspect of these computers was their software architecture. Not only did they support graphical interaction, but they also provided a software toolkit of reusable interactive components. In 1984, MIT formed Project Athena, which developed the X Windows system. Project Athena also provided a toolkit of reusable pieces of interaction called “widgets”². The Athena widgets provided reusable interaction for the UNIX world. In this chapter we will explore how widgets work and how they simplify the creation of graphical user interfaces.

There are many different widget toolkits available in the world. Some are built around operating systems such as Microsoft Windows or Apple’s OS 10. Many are built around the unique capabilities of languages such as Java’s Swing³ toolkit or the components that come with C#⁴. There are also toolkits for specialized hardware such as personal digital assistants (PDAs)⁵ or tablet computers. Despite this diversity there are a number of ideas that are common across almost all of these systems. This chapter discusses these general ideas. Between the time of writing this chapter and its publication, there will be at least one new widget toolkit introduced by someone for some reason. Therefore we will focus on the central ideas and leave it up to the reader to learn the specifics of a particular implementation.

Pre-built widgets fall into three categories: simple widgets, container widgets and abstract model widgets. We will discuss the first two categories in this chapter and leave the abstract model widgets for chapter 7. The simple widgets are a modification of the model-view-controller architecture where the model,

view and controller are all packaged in a single class as shown in figure 4.1. The reason for this is that the model is not sufficiently complex to justify a separate class. The models for most widgets are single values, such as a number, text string, boolean value or list of choices. The advantage of such simple widgets is that the programmer does not need to implement them, their look and feel becomes standardized across all applications and most of their design choices are based on changing properties such as colors, fonts, sizes. Interface design based on changing properties rather than writing code makes it possible to create the interface design tools we will discuss in chapter 9.

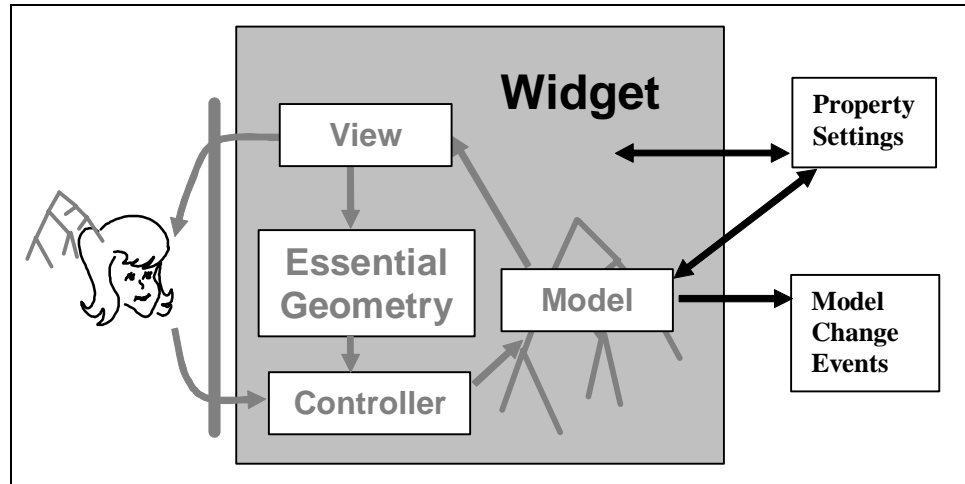


Figure 4.1 – Widget architecture

When using a widget, we must give it a position on the screen, set its properties and arrange to bind its model change events to the appropriate application code. We will discuss the layout problem (positioning widgets) in chapter 5. The model change events historically were handled using the callback strategy. This allowed textual callback names to be specified as part of a widget's properties. This transitioned to object inheritance model and quickly outgrew that model with the number of events to be handled and the number of classes that must be declared to handle those events. Most modern widget toolkits use either the listener or delegate models described in chapter 3. Some interactive design tools such as forms layout tools in database systems or web design tools will use the interpretive language or reflection event handling strategies.

Figure 4.2 shows a dialog box design consisting of a variety of widgets. These widgets are arranged in the usual tree. At the top level there is a container

widget that holds the “General” tab, the “Print” and the “Cancel” buttons. The “General” tab has three container widgets: the “Select Printer” group, the “Page Range” group and the “Copies” group. Each of these then contains other widgets. The widget tree reflects the window tree. Each widget has its own window to manage and provides it own view and controller to interact through that window.

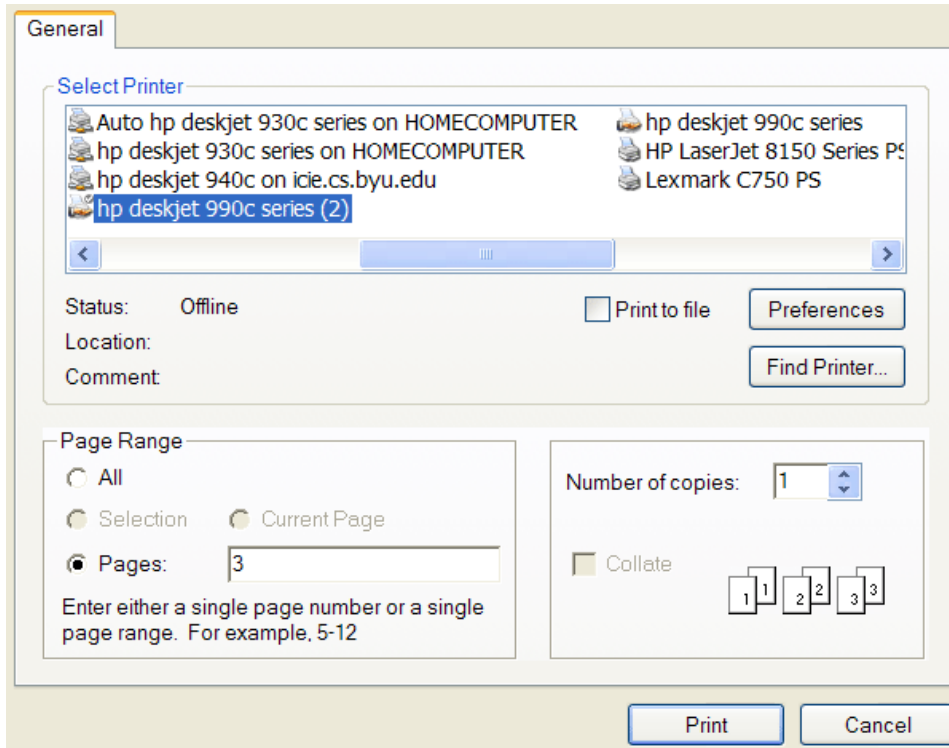


Figure 4.2 – A dialog built from widgets

The setting of widget properties is fundamental to widget-based interface design. The most important property that is common to all widgets is the Bounds. This is the rectangular area that the widget is to use as its window. The coordinates of the Bounds are almost always defined relative to the coordinates of the widget’s container. This allows a container to be moved around while all of its children remain in their relative positions within the container.

There are two styles for getting and setting properties. These depend on whether the programming language supports field accessors or not. Java does not have field accessors. Therefore we might use two separate methods and the

set/get naming convention shown in figure 4.3. This pattern of using two methods with the get/set naming convention is used by interface design tools to discover the properties of a widget. We need to ensure property access through methods because when a property is changed, the view must be notified. This notification is internal to the widget class, but it still must be handled. In the case of changing the Bounds, the layout algorithm (Chapter 5) must be invoked.

```
public class myNewWidget extends Widget
{
    private Rectangle myBounds;
    public Rectangle getBounds() { return myBounds; }
    public void setBounds(Rectangle newBounds)
    {
        . . . view notification code . . .
        myBounds=newBounds;
    }
}
```

Figure 4.3 – Get/Set accessor methods

Some languages, such as C#, provide explicitly for accessors to fields. Figure 4.4 shows an example of such field accessors. This provides field-like access (`myWidget.bounds`) with underlying methods to control the access. This also simplifies the reflection code that must discover such properties in the interface design tools.

```
public class myNewWidget extends Widget
{
    private Rectangle myBounds;
    public Rectangle bounds
    {
        set {      . . . view notification code . . .
                myBounds=value;
        }
        get {      return myBounds; }
    }
}
```

Figure 4.4 – C# field accessors

In the sections that follow we will look at many kinds of widgets. Generally such widgets are characterized by:

- the model information that they manipulate,
- the properties that the programmer/designer can set to control the behavior or appearance, and
- the events that the widget generates.

We will first look at simple widgets that have a single value as their model and provide the basis of any widget tool-kit. Next we will look at container widgets. These provide mechanisms for grouping other widgets together to create more complex user interfaces. An important class of widgets are those created for a particular application. These form the heart of most user interface implementation efforts. Lastly we will look at how of these widgets can be collected together to form a higher-level model-view-controller architecture for our application.

Simple Widgets

An early concept in the development of interactive systems was the notion of *logical input devices*. A logical input device is defined by its function and its interface. The actual implementation on the screen is separate. This is an important design concept because it supports exploring various styles of interaction without changing the way in which they interact with the underlying model. Rapid interchange of interactive style is very important to the process of creating interactive designs that work effectively. For example, we can define the logical device “button” that simply generates an event when it is pressed. Figure 4.5 shows several possible implementations of a logical button. A single button concept can be an icon on a tool bar, a physical keyboard button, a menu item, a control sequence such as “Ctrl+P”, or a picture of a button on a screen. Though each is implemented very differently they all have the same relationship to the model, which is “do this thing”. Most widget sets are organized around the model for each widget. For each kind of model there is a corresponding widget. Using such a widget set application models can be represented interactively.



Figure 4.5 – Implementations of logical buttons

The button widgets shown in figure 4.5 do not really have a model. Their purpose is to generate events. In most toolkits all of these types of buttons generate the same `ActionEvent`. This makes the application software independent of the kind of button that produced the event.

The simple widgets are grouped by the models that they handle. We will look at information or style widgets used to dress up our interfaces. These are followed by boolean and choice widgets that provide selections. Text widgets are an important group for a variety of purposes. There are also many variations of widgets for interacting with numbers. Lastly we will look at several special-purpose widgets that are found in most toolkits.

Information and Style Widgets

The simplest widgets are those that provide information or style. The most notable examples are labels and images. A label provides a piece of text that does not interact in any way. It just shows its text. Image widgets similarly show whatever image is associated with them. Image widgets are very useful in creating widely varying styles to attract attention or to set a mood. For example, in figure 4.6 we see two different “skins” for the Windows Media Player. The logical behavior of all of these “skins” is the same, the stylistic differences are provided by the images that some artist has drawn into which the various control widgets have been embedded. The usual properties for style or information widgets are label text, images, font, text color and background color.



Figure 4.6 – Varying media player “skins”

Boolean Widgets

Most widgets display and modify some value. The simplest are the boolean widgets. The most common forms are the checkbox and the toggle button shown in figure 4.7.

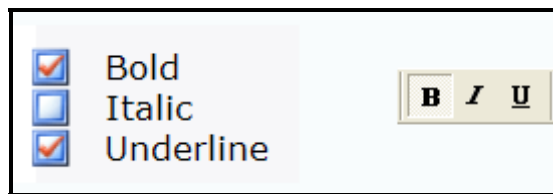


Figure 4.7 – Boolean widgets

There is some variation in how check boxes are organized. In some systems there is a label associated with the check box. In other systems the check box is implemented by itself with a label widget being placed next to it to provide the information. The two-widget approach simplifies the implementation while providing a lot of flexibility. However, in practice designers do not vary much from the standard box/label style. Packaging them together is simpler for most programmers. When a checkbox includes a label there is frequently a property to define where the label should be placed.

Theses generate an event whenever their model changes state. They share most of the properties of the label widgets. In some toolkits there is a choice of styles for the state indicator. In other toolkits there are `trueImage` and `falseImage` properties that allow the designer to provide any two images to represent the state of the widget. These images can be applied to the entire button as on the right or

just to the indicator box as on the left. The use of images allows for wide stylistic variation without changing the implementation.

Choice from a list

A very common and highly flexible interactive technique is to select one or more choices from a list of alternatives. There are several issues that influence the interactive style of such widgets. Is it possible to select one choice or many? The most common case is one. Are there a relatively few choices or many choices? Is the list of choices dynamic and can change with time? Figure 4.8 shows a variety of choice widgets.

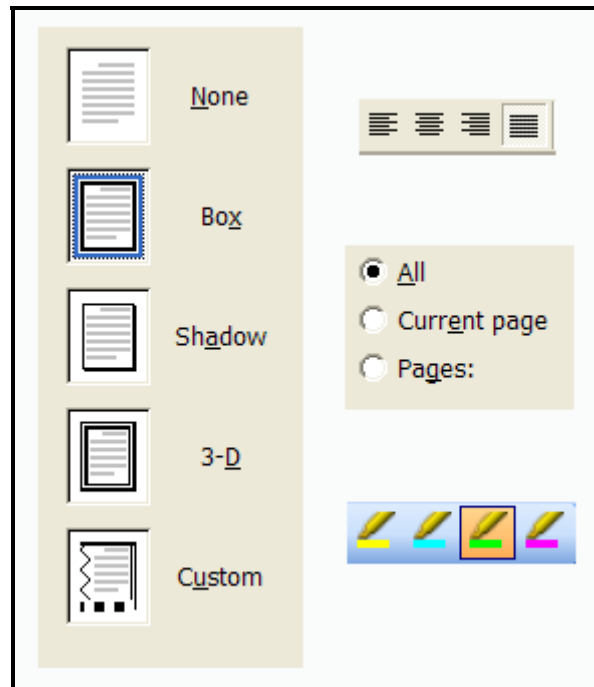


Figure 4.8 - Fixed choice widgets

All of the widgets in figure 4.8 are implemented using a standard technique. Each choice has two images (selected/not selected) and an optional text label. Again the image/label technique pushes widget design into the realm of artists rather than programmers. The classic radio button style in the middle right was made popular by the Macintosh. Historically this style of choice is called a radio-

button because of its similarity to the station selector buttons on car radios. This style is used when the number of choices is small and fixed.

Conceptually the model for a choice widget is a single variable that can take on multiple values. Depending on the system, these choice values are usually strings or integers. A single model value is shared among the choice widgets in a group. Each choice has a specific constant value associated with it. When the shared variable has the same value as a choice widget, that choice is shown as selected, otherwise it shows as not selected.

A group of single choices is generally not treated as a single widget. Rather each button is created as a widget of its own and then groups of them are linked together. This strategy allows for wide variations in layout of the choices including columns, rows, tables or others. These individual widgets must be grouped together in mutually exclusive groups. In JavaScript this is done by having several widgets share the same name. In C# all radio buttons placed on a Form are mutually exclusive. Each button has a checked property that specifies which of the buttons is the selected one. There is also a special container widget called a `GroupBox` that handles mutual exclusion for radio buttons on a finer scale than the whole window. In Java there is a special class called `ButtonGroup`. The `ButtonGroup` manages the shared variable that controls which of its members is selected. Unlike `GroupBox`, a `ButtonGroup` is not a widget. It is an invisible object that ties together radio buttons.

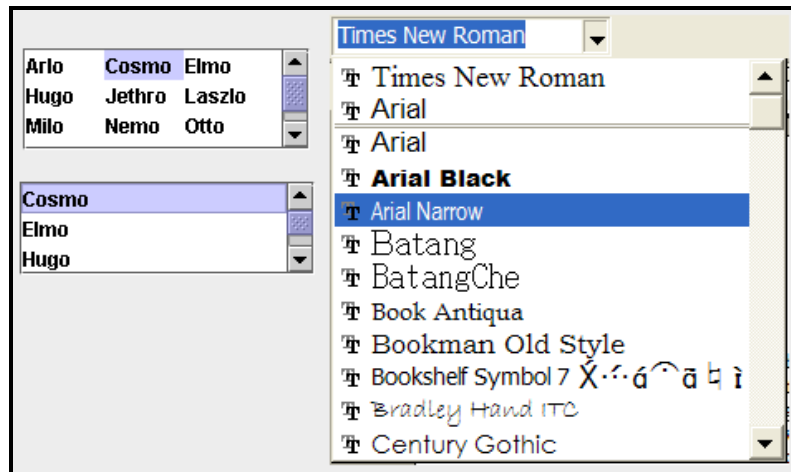


Figure 4.9 – Large choice lists

The widgets shown in figure 4.9 have the ability to handle more choices than can fit on the screen at one time. The widget on the right is a Microsoft ComboBox. The two on the left are variants of Java/Swing's JList widget. These widgets are also special because the program can dynamically change the set of choices.

The properties of choice widgets include the appearance properties found on labels. There may be images to indicate choice and they may be properties to indicate how choices should be laid out. When many choices are possible there will be a property indicating how many should be shown at once. This sets the size of the scroll region for the widgets in figure 4.9.

The model for choice widgets has a list of choices and a selected item or items. Selection is represented as an index into the list of choices. Such choice lists can frequently handle multiple selections. Some implementations allow for a range of selections represented with a start and end index. Others allow for multiple ranges. The most common technique, however, is the single selection.

Text

The text box is one of the most commonly used widgets. Its model, of course, is a string of text. In addition, some implementations expose the start and end of the selection when the user has selected some region of the text in the box. Text boxes come in multi-line and single line variants. In Java/Swing the `TextField` is single line and the `TextArea` is multi-line. In C# the `TextBox` has an attribute that can switch the same widget between multi-line and single-line modes. Multi-line text boxes also have properties that control word wrapping and alignment.

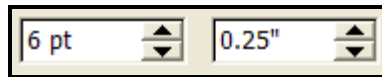


Figure 4.10 – Formatted text boxes

In many cases we use text boxes to represent numbers or numbers with specific units as shown in figure 4.10. There are a wide variety of ways in which text should be restricted in its format. To support this, some such widgets provide a *formatter* object that can be set as a property of such text boxes. When the user changes the text in the box, the formatter is sent the new string. The formatter then accepts the string, rejects the string or modifies it to conform. Most toolkits that support formatted text provide a wide variety of pre-built formatter classes ranging from percentages, decimals, measurements, dates, times and IP addresses.

Some toolkits provide text boxes that can handle formatted text such as bold, italics, and font changes. Java/Swing provides the `JEditorPane` that can edit HTML. C# provides much of the functionality of the Internet Explorer web browser as a widget.

Text widgets generate events when the text changes. In addition they will generate events when the selection changes. For example an application could provide a dictionary widget that shows the definition of any selected word. This dictionary could listen to selection change events from a text box and change its definition whenever the selected text changes.

Number widgets

There are a variety of widgets for specifying or displaying numbers. Figure 4.11 shows several such widgets. The number widgets generally have a maximum, minimum and current value. The user controls the current value. The properties of such widgets control the color, font and sometimes the orientation (vertical/horizontal). These widgets generate events whenever the number they control is changed. There are some differences as to how often change events are generated. One style will generate an event every time the slider moves. Others only generate a change event when the drag of the slider is complete. Generating an event on every change allows other information to change continuously such as the text box to the right of the bottom widget in figure 4.11. However, this generates many events. Some widget sets provide a property to control when change events are generated. Others provide two different events: one for incremental changes and another for the final change.

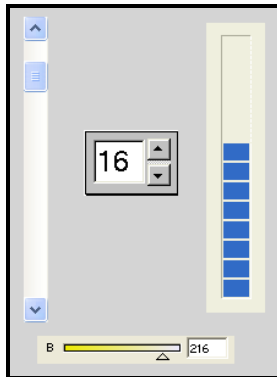


Figure 4.11 – Number widgets

Special Values

There are a variety of special values for which widgets have been developed. Some of these are provided as built-in parts of a tool kit, others are developed for special purposes. Common built-in widgets include: a color picker (figure 4.12), selecting where to save a file (figure 4.13), selecting a date from a calendar (figure 4.14) and selecting a time of day (figure 4.15).

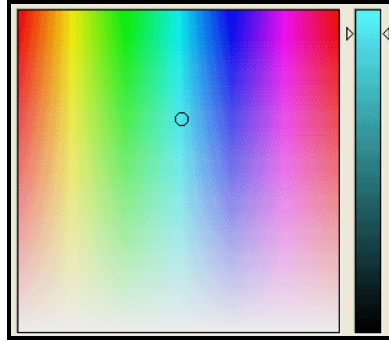


Figure 4.12 – Color selection widget

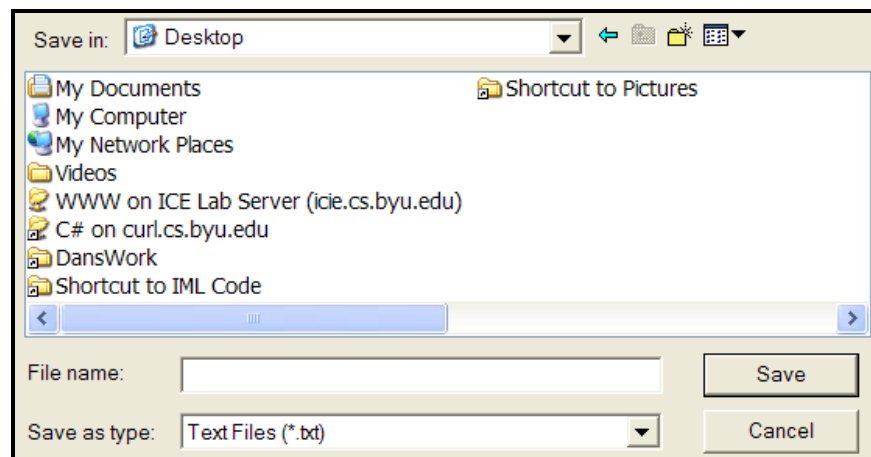


Figure 4.13 – File save widget

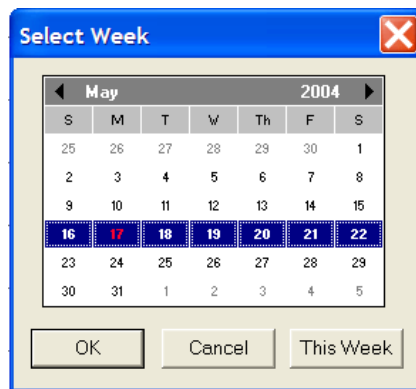


Figure 4.14 – Calendar widget

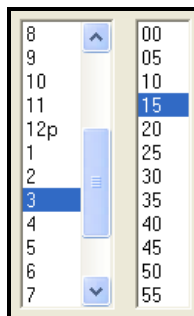


Figure 4.15 – Time selection widget

Many of these widgets are used in a *modal* fashion. A modal dialog is one where only a particular kind of interaction is possible. For example when trying to select a color, all other parts of an application are disabled and will not accept input. The reason for this is control of the dialog. The user may have specified that the foreground color should be changed. The dialog cannot proceed until the user specifies the new color. In situations where the application needs a value in order to proceed, a modal dialog is required. However, we could embed the color selection widget in the interface so that it is always visible. Whenever the user changes the selected color, the model is updated appropriately. This would be a *modeless* dialog because the user is not required to change the color before proceeding. This can be freely done at any time. However, if the application has foreground color, background color and text color, putting all three color pickers on the screen would be a waste of screen space. Instead we put color icons that show the current selection for each such color and then launch a modal dialog whenever the user specifies a change. In most cases the switch from modeless

(preferred) to modal (more restrictive) is dictated by the need to conserve screen space.

Container Widgets

The widgets that we have discussed so far all support the editing of simple values of words, strings, numbers, colors, dates etc. To build realistic interfaces we need to collect such simple values together to form accommodate more complex models. The container widgets serve the same function in the user interface as a struct in C or an object with member fields in Java. There are a variety of such container widgets. A container widget's model is a list of other widgets. Container widgets generally do not generate events. That is delegated to the widgets that actually have values. At the end of this chapter we will show how container widgets can serve as collectors of events and provide controllers at a higher level of abstraction. The common container widgets are menus, forms, tabs and toolbars.

Menus

A menu is one of the simplest widget collections. All of the items in a menu are stacked up either vertically or horizontally as shown in figure 4.16. For user interface consistency, menus are handled in the same way across a given operating system. This is so that users always know how to find the commands that they need for any new application. Generally menus are composed of buttons that perform actions, open modal dialogs or open other menus. Sometimes check boxes or radio buttons are found in menus.

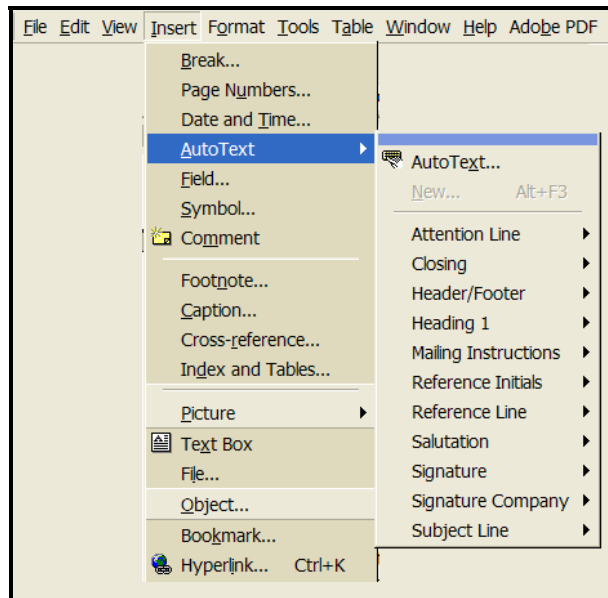


Figure 4.16 – Menu container

Panes/Forms

Another common form of container widget is the pane or form. The term pane comes from the panes in a window. Widgets are laid out on a 2D surface just like a paper form as shown in figure 4.17. Handling widget layout in a pane can become complex when the user can change the size of the pane. We will discuss these layout questions in more detail chapter 5.

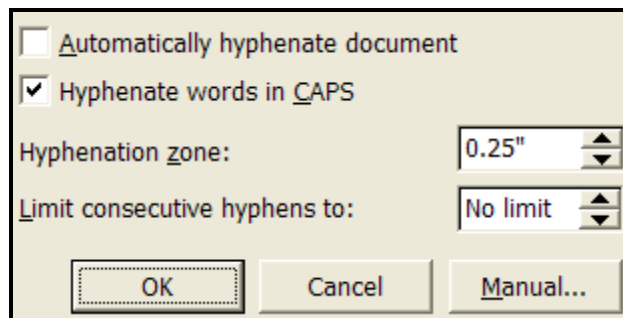


Figure 4.17 – Simple pane of widgets

Panes can also be nested within other panes as shown in figure 4.18. This figure also shows that panes can have borders with labels for the group of widgets collected together by the pane. Each pane has a collection of widgets that can be any possible widget including other panes. This provides us with trees of widgets and supports tree models where the shape of the tree is fixed. For more dynamic trees we will use the higher order widgets described in chapter 7.

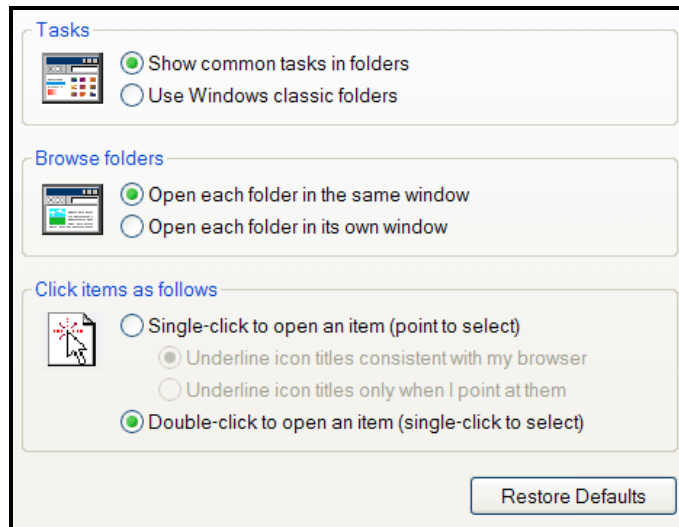


Figure 4.18 – Nested panes

Tabs

Collecting widgets together in panes can take up a lot of screen space. There are frequently more controls than can possibly fit. One solution to the screen space problem is to use a tab container. Just as with a pane, a tab container widget contains several child widgets, each with an associated name. The user is provided with a set of tabs to select which widget should be visible at any time. Figure 4.19 shows a simple row of tabs for selecting widgets. Figure 4.20 shows multiple rows of tabs where there are many choices.

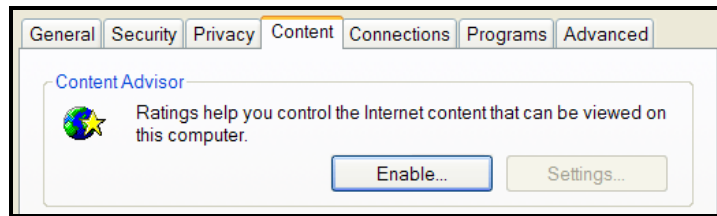


Figure 4.19 – Simple tabbed pane

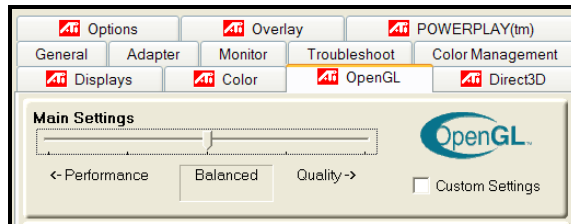


Figure 4.20 – Multi-row tabs

Toolbars

Another common container for simple widgets is the tool bar. It has the advantage of arranging a variety of simple widgets along the top or bottom of a window in a way that is easy to access and yet takes up very little screen space, as shown in figure 4.21. The success of a toolbar depends upon having widgets that are frequently used and whose purpose can be conveyed in very little screen space.



Figure 4.21 – Toolbar

Application Widgets

Beyond the built-in widgets provided with most user interface toolkits there are almost always widgets that the programmer must create to perform application-specific tasks. For example, a word processor would have many checkbox, button, combo-box and text widgets around the periphery of the window. However, most of the window will be occupied by the document and its text. This interaction is the heart of the word processor and must be implemented especially for this application. Almost all interactive programs have one or more key widgets that are specially constructed for that application. The

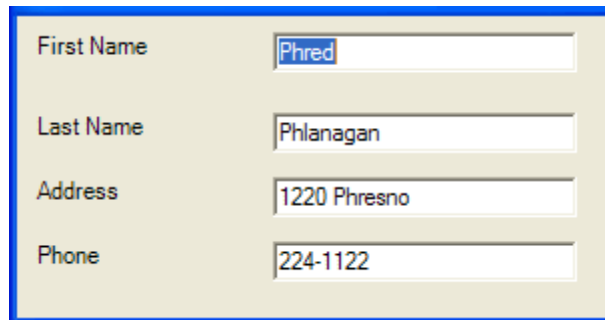
implementation of these widgets use the techniques discussed in chapters 2 and 3.

Model-view-controller with widgets

Though each simple widget has its own model, view and controller we need to map these to the needs of our application. To create an application using preexisting widgets we must address the following issues.

- Build a model for the user interface with a change notification/listener mechanism
- Create the tree of widgets (both prebuilt and application specific) that will define the view.
- Implement the view code to handle model change notifications and change widget properties so that the new information will appear.
- Create a controller by attaching listeners to any prebuilt widgets so that changes made by the user can be propagated to the model. This controller gets its events from other widgets rather than directly from the input devices.
- Resolve notification race conditions.

Suppose we have a simple application for managing member lists for our soccer team. Each player would have information like that shown in figure 4.22.



First Name	Phred
Last Name	Phlanagan
Address	1220 Phresno
Phone	224-1122

Figure 4.22 – Team member form

The model for the user interface in figure 4.22 is the Java class shown in figure 4.23. As with our previous models this one must implement a notification

mechanism so that the view can determine when the model has changed. Because this is a rather simple model we will use the listener interface shown in figure 4.24.

```
public class TeamMember
{
    public String getFirstName() { ... }
    public void setFirstName(String firstName) { ... }
    public String getLastName() { ... }
    public void setLastName(String lastName) { ... }
    public String getAddress() { ... }
    public void setAddress(String address) { ... }
    public String getPhone() { ... }
    public void setPhone(String phone) { ... }

    public void addTeamMemberListener(TeamMemberListener listener) { ... }
    public void removeTeamMemberListener(TeamMemberListener listener) { ... }
}
```

Figure 4.23 – Java model for team member

```
public interface TeamMemberListener
{
    public void changed();
}
```

Figure 4.24 – TeamMemberListener

A view for our team member model is a panel container that contains four panel containers. Each of the four contains a label and a text box. The implementation of the view below is simplified to ignore layout specifications. We will discuss layout in chapter 5. In this case `TeamMemberView` extends `JPanel`, which is a Java/Swing container widget. By extending the container we can set up our other widgets as well as the event handling. Figure 4.25 shows how we might set up our `TeamMemberView` widgets. The setup all occurs in the constructors. The `TeamMemberView`'s constructor creates `LabeledText` widgets and adds them to itself. `JPanel` does not know about any widgets that it contains until they are added. The `add()` method adds widgets into the widget tree where they can be drawn and receive events. Just creating a widget is not enough. It must be added into the widget tree. `TeamMemberView` also saves each of the `LabeledText` widgets in a local variable where we can refer to them later. There is one for each piece of model information.

```

public class TeamMemberView extends JPanel implements TeamMemberListener
{
    private TeamMember myModel;

    private LabeledText firstName;
    private LabeledText lastName;
    private LabeledText address;
    private LabeledText phone;
    public TeamMemberView(TeamMember model)
    {
        myModel=model;
        myModel.addTeamMemberListener(this);

        firstName=new LabeledText("First Name");
        this.add(firstName);
        lastName=new LabeledText("Last Name");
        this.add(lastName);
        address=new LabeledText("Address");
        this.add(address);
        phone=new LabeledText("Phone");
        this.add(phone);
    }
    private class LabeledText extends JPanel
    {
        private JTextField textField;
        public LabeledText(String label)
        {
            this.add(new JLabel(label));
            textField = new JTextField();
            this.add(textField);
        }
    }
}

```

Figure 4.25 – Widget setup

The `LabeledText` is a private class that is implemented as another subclass of `JPanel`. In its constructor we add a label widget, with the appropriate label text and a `JTextField` widget that will contain the information. We remember the `JTextField` in a private member for use later.

Figure 4.25 has set up the view. What we do not have is the model change notification nor the controller. Figure 4.26 shows `TeamMemberView` with the change notification added. `TeamMemberView` implements the `changed()` method required by `TeamMemberListener`. In this method it extracts information from the model and then notifies each `LabeledText` widget that its text has changed by calling `LabeledText.setText()`. The `LabeledText` widget in turn calls `setText()` on its own `textField` widget. Java/Swing in its `JTextField` implementation will call `damage()` so that the windowing system knows that the text will be redrawn. `JTextField` also implements

the `redraw()`. All our code needs to do is to give the new string the text field widget.

```
public class TeamMemberView extends JPanel implements TeamMemberListener
{
    private TeamMember myModel;

    private LabeledText firstName;
    private LabeledText lastName;
    private LabeledText address;
    private LabeledText phone;
    public TeamMemberView(TeamMember model)
    { ... }
    private class LabeledText extends JPanel
    {
        private JTextField textField;
        public LabeledText(String label)
        { ... }
        public void setText(String newText)
        { textField.setText(newText); }
    }
    public void changed()
    {
        firstName.setText(myModel.getFirstName());
        lastName.setText(myModel.getLastName());
        address.setText(myModel.getAddress());
        phone.setText(myModel.getPhone());
    }
}
```

Figure 4.26 – Model change notification

We now need to add controller code to our `TeamMemberView`. However, we will not be handling mouse events. That will be done by the pre-built widgets. We need to handle the events produced when the user types new text into a text box. In Java/Swing the `JTextBox` will produce an `ActionEvent` whenever its text is changed. We want our `TeamMemberView` to receive such events so we have it implement the `ActionListener` interface as shown in figure 4.27. In `TeamMemberView`'s constructor we add it as a listener to each `LabeledText` widget. The `LabeledText` widget in turn adds the listener to its `textField` box, since that is the widget that will generate the events.

```

public class TeamMemberView extends JPanel
    implements TeamMemberListener, ActionListener
{
    private TeamMember myModel;

    private LabeledText firstName;
    private LabeledText lastName;
    private LabeledText address;
    private LabeledText phone;
    public TeamMemberView(TeamMember model)
    {
        ... other setup code ...
        firstName.addListener(this);
        lastName.addListener(this);
        address.addListener(this);
        phone.addListener(this);
    }
    private class LabeledText extends JPanel
    {
        private JTextField textField;
        ... other methods ...
        public void addListener(ActionListener listener)
        {
            textField.addActionListener(listener);
        }
        public String getText()
        {
            return textField.getText();
        }
    }
    ... other methods ...
    public void actionPerformed(ActionEvent e)
    {
        myModel.setFirstName(firstName.getText());
        myModel.setLastName(lastName.getText());
        myModel.setAddress(address.getText());
        myModel.setPhone(phone.getText());
    }
}

```

Figure 4.27 – Listening to events

The `ActionListener` interface requires an `actionPerformed()` method that will be called whenever the user changes text in a `JTextField`. The `actionPerformed()` method gets the text from each `LabeledText` widget (who gets it from their `textField`) and uses that text to set the various fields on the model. Thus whenever the user enters any text, the action event is generated. Since `TeamMemberView` is a listener on all of those events, its `actionPerformed()` method is called to get the information out of the text widgets and send it to the model.

This is a good point to review the various event handling strategies discussed in chapter 3. If we were using the inheritance event handling strategy, then we would need to make subclasses of `JTextField` to handle the action events. This is awkward for two reasons. The first is that we do not want to make all of those

subclasses and the second is that it is `TeamMemberView` that wants to handle the events because `TeamMemberView` has the model. The listener event mechanism used in figure 4.27 allows the `TeamMemberView` (which is not in any way related to the implementation of `JTextField`) to receive and handle the events in a way that is appropriate to the model.

This example also shows the deficiencies of the listener model. Any change to any `JTextField` will generate the same call to `actionPerformed()`. In the `ActionEvent` parameter we could find out which text box generated the event but they all come through the same `actionPerformed()` method. What we would like to do is have a separate event method associated with `firstName`, `lastName`, `address`, and `phone`. Each of these methods would change only the model field that needed to be changed. The delegate event handling strategy found in C# and other languages would do exactly this.

Notification race conditions

There is one final problem with notification race conditions. If the user enters some text in the “Last Name” field, an `actionPerformed()` will be called on our view. The `actionPerformed()` will call `setFirstName()` on the model (figure 4.27), which in turn will notify all views listening to the model that the model has changed. Because our view is listening to the model, its `changed()` method (figure 4.27) is called. In the `changed()` method we call `setText()` which will change the text in the `JTextField`. In many widget implementations events are generated whenever the widget model changes, no matter who changed the model. `JTextField` is one such widget. Changing its text will generate an `actionPerformed()`, which will change the model, which will notify the view, which will change the text, which will generate `actionPerformed()`, and so on.

These race conditions happen frequently in user interfaces because they are inherently a loop of mutual notifications. We need to be aware of these possibilities and where necessary block the loop. Figure 4.28 shows one mechanism to handle the problem. A private flag `activeAction` is initialized to false. This will indicate when we are already handling an action event. If `actionPerformed()` is called when it has already been called previously, it will simply return and block the loop. If it is not currently active, it sets `activeAction`, does its business and then clears the flag. It is very important to understand the conditions under which prebuilt widgets will generate events. Frequently this is not documented clearly and can only be discovered by experimentation with the code.

```

public class TeamMemberView extends JPanel
    implements TeamMemberListener, ActionListener
{
    private boolean activeAction;
    public TeamMemberView(TeamMember model)
    {
        ... other setup code ...
        activeAction=false;
    }
    private class LabeledText extends JPanel
    {
        ...
    }
    ... other methods ...
    public void actionPerformed(ActionEvent e)
    {
        if (activeAction) return;
        activeAction=true;
        myModel.setFirstName(firstName.getText());
        myModel.setLastName(lastName.getText());
        myModel.setAddress(address.getText());
        myModel.setPhone(phone.getText());
        activeAction=false;
    }
}

```

Figure 4.28 – Blocking race conditions

Summary of Widgets

Prebuilt widgets greatly simplify the development of user interfaces by standardizing the look and feel of the interface and eliminating the need to write and debug relatively complex code. Widgets also move much of the user interface design problem from coding of graphics and mouse events to the selection and setting of properties.

With the exception of the information and style widgets, all prebuilt widgets generate events. It is normal to have some container widget collect those events and handle the interface between the view and the model. The events from the simple widgets are translated into changes to the model. Model change notifications are translated by the view into changes to widget model properties.

Various toolkits will use one or more of the event handling strategies described in chapter 3. Any of the object-oriented techniques are in wide use. The delegate strategy is the simplest for stitching widgets together into more complex applications. Events generated by prebuilt widgets can produce notification race conditions that must be handled by the programmer.

¹ Bewley, W. L., Roberts, T. L., Schroit, D., and Verplank, W. L. "Human Factors Testing in the Design of Xerox's 8010 "Star" Office Workstation,"

Human Factors in Computing Systems (CHI '83) ACM Press, New York, NY, (1983) 72-77.

² McCormack, J. and Asente, P. "An Overview of the X Toolkit," *Symposium on User interface Software (UIST '88)*, ACM Press, New York, NY, (1988) 46-55.

³ Zukowski, J., *The Definitive Guide to Java Swing*, Apress (2005).

⁴ MacDonald, M., *User Interfaces in C#: Windows Forms and Custom Controls*, Apress (2002).

⁵ McKeehan, J. and Rhodes, N., *Palm OS Programming: The Developer's Guide*, O'Reilly Media, (2001).