Extracted from:

FXRuby

Create Lean and Mean GUIs with Ruby

Chapter 8

Building Simple Widgets

Widgets are the building blocks of GUI applications. They are special-purpose objects that are displayed onscreen and can be manipulated to allow communication between users and software. If you're like most people, a majority of the computer software that you work with on a daily basis incorporates some kind of graphical user interface, so working with widgets is second-nature to you even if you don't consciously think of it in those terms. When you click a drop-down menu in your word processor and choose a command from that menu, you're interacting with the application through widgets. When you grab the scroll bar on the right side of the document pane and drag it up and down to scroll back and forth through the document, you're again using a widget to interact with the software.

In his book *About Face* [Coo95], Alan Cooper talks about the four basic types of widgets, as evaluated in terms of users' goals:

- Imperative widgets, which are used to initiate a function
- Selection widgets, used to select options or data
- Entry widgets, used to enter data
- Display widgets, used to directly manipulate the program visually

Some widgets can of course meet combinations of these goals. Figure 8.1, on the next page is a list of the widgets we'll cover in this chapter, along with advice about when to use them. We don't have enough space in this book to describe all the widgets provided by FXRuby, but if you study some of the most commonly used widgets, you'll pick up on the terminology and naming conventions that are used throughout the library. After you finish reading this chapter and the next few chapters, you will have the skills you need to integrate other more specialized widgets into your application with no problem.

Widget Class	What's It For?
FXLabel	Use a label to display text, with an optional icon, for decorative or informative purposes.
FXButton	Use a button as a "pushable" interface to an imperative command.
FXRadioButton	Use a group of radio buttons when you need the user to select one from many possible options.
FXCheckButton	Use a check button to allow the user to select or deselect an option.
FXTextField	Use a text field to allow the user to edit a single line of text.
FXToolTip	Use a tooltip to display a temporary, informative message about the purpose of some other widget.
FXStatusBar	Use a status bar to display detailed, context-sensitive help about the purpose of some other widget or the state of the application.

Figure 8.1: Simple Widgets

8.1 Creating Labels and Buttons

I don't think I've ever developed a GUI application that didn't have at least a few labels and buttons. They're easy to understand and simple to use, so this seems like a good place to start.

Displaying Text with Labels

We can use an FXLabel widget to display a message on a user interface. It can be simple, such as a title string for a feature in the user interface, or more complicated, such as a set of instructions for some task. The label text can consist of one or more lines, separated by newline characters, and the label can optionally display an icon. Technically speaking, the text for a label is optional too, and it's possible to display a "label" widget that has only an icon, but this practice is more common for button widgets (which we'll cover in the next section).

By default, the label text is centered (both horizontally and vertically) inside the label's bounding box. However, the label supports a number of different justification options that we can use to specify how the

Figure 8.2: Label displaying left-justified text

Figure 8.3: Label displaying bottom- and right-justified text

label's text is aligned. For example, to left-justify the label text, pass in the JUSTIFY_LEFT option to the FXLabel constructor.

Download labelexample1.rb

```
label = FXLabel.new(self, "Left-justified text", :opts => JUSTIFY_LEFT)
```

Figure 8.2 shows how this label's text is displayed left-justified. You can also modify the text justification for a label by setting its justify attribute.

Download labelexample2.rb

```
label.justify = JUSTIFY_RIGHT|JUSTIFY_BOTTOM
```

Figure 8.3 shows the label text, justified against the right and bottom sides of the label's bounding box. For a listing of the text justification options, see the API documentation for the FXLabel class.

By default, a label is drawn without any kind of frame around it. The frame style can be changed by passing in a combination of frame-style flags to the FXLabel constructor. For example, to create a label with a solid line around its border, use the FRAME_LINE frame style.

Download labelexample3.rb

```
line_frame = FXLabel.new(p, "Line Frame", :opts => FRAME_LINE)
```

Figure 8.4: Labels displaying the various frame styles

You can also change the frame style by setting the label's frameStyle attribute after it has been created. In this example, we're giving the label a sunken frame.

Download labelexample3.rb

```
sunken_framed_label = FXLabel.new(p, "Sunken Frame")
sunken_framed_label.frameStyle = FRAME_SUNKEN
```

Figure 8.4 shows examples of labels with all the supported frame styles. For a complete listing of the available frame styles, see the API documentation for the FXWindow class.[1]

As I mentioned earlier, you can also include an icon with a label. For example, you can construct an icon from a GIF format file and then pass it in as an argument to the FXLabel constructor.

Download labelexample4.rb

```
question_icon =
  FXGIFIcon.new(app, File.open("question.gif", "rb").read)
question_label =
  FXLabel.new(self, "Is it safe?", :icon => question_icon)
```

1. The frame style constants are associated with the FXWindow base class because a lot of different kinds of widgets have associated frame styles, not just labels.

Figure 8.5: A label with an icon

We'll go into more detail on the topic of how to create icons in Chapter 11, *Creating Visually Rich User Interfaces*, on page 144. For now, it's enough to know that you can construct an icon object from an image file (or some other source) and then use it as decoration for labels, buttons, and several other kinds of widgets.

By default, the icon will appear centered inside the label's bounding box, which could make the label text difficult to read unless the icon is transparent or there's high contrast between the icon's colors and the text color. The label supports a number of options that we can use to specify how the text and icon are positioned with respect to one another. For example, to place the icon before the text (in other words, to its left), use the ICON_BEFORE_TEXT option.

Download labelexample4.rb

```
question_label.iconPosition = ICON_BEFORE_TEXT
```

Figure 8.5 shows how this example looks running under Windows. Again, refer to the API documentation for the FXLabel class for a complete listing of the available icon position settings.

Getting Pushy with Buttons

An FXButton is a step up from a label in the sense that it can be "pushed" and that, when pushed, it executes some command in your program. Like a label, a button can display a message string and an icon. Unlike a label, a button typically has a 3-D "raised" appearance that makes it stand out. Figure 8.6, on the next page, shows two different buttons, one enabled and one disabled.

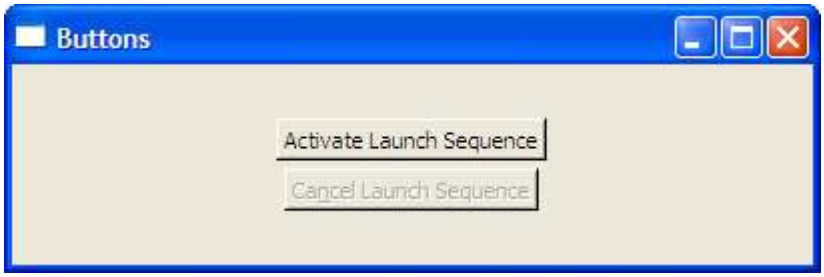

Figure 8.6: Unlike labels, buttons look “pressable.”

When a button is pressed and released, it sends a SEL_COMMAND message to its target:

Download buttonexample.rb

```
activate_button = FXButton.new(p, "Activate Launch Sequence",
  :opts => BUTTON_NORMAL|LAYOUT_CENTER_X)
activate_button.connect(SEL_COMMAND) do |sender, sel, data|
  @controller.activate_launch_sequence
end
```

A button will also send a few other messages to its target, but those messages are rarely useful in practice. If you’re interested in reading about those, remember that you can always refer to the online API documentation for a class to learn about all the messages it sends to its target.

You can associate accelerators and hotkeys with buttons and other kinds of widgets. An *accelerator* is a combination of keystrokes that invokes an action in your application. For example, pressing Ctrl+C will invoke the Copy action in many applications. A *hotkey* is a special kind of accelerator that is a combination of the Alt key and a letter, such as Alt+F to open the File menu.

Hotkeys are most commonly associated with buttons, or buttonlike widgets, that have an associated text string. You encode the hotkey for the button by doing this:

Download buttonexample.rb

```
cancel_button = FXButton.new(p, "Ca&ncel Launch Sequence",
  :opts => BUTTON_NORMAL|LAYOUT_CENTER_X)
```

In this example, the ampersand character that precedes the letter *n* in the button's label indicates that the Alt+N keystroke should trigger this button's command, just as if the user had clicked this button with the mouse.

We talked in the previous chapter about how to use the automatic GUI updating feature to update the state of the user interface widgets depending on the application state. There's a fairly common situation in which you might want to use this to change the state of a button, and it's when you want to disable a button because the command associated with that button isn't currently available. For example, we should disable the Cancel Launch Sequence button if the launch sequence hasn't been activated. You can write a SEL_UPDATE handler to account for this:

Download buttonexample.rb

```
cancel_button.connect(SEL_UPDATE) do |sender, sel, data|
  sender.enabled = @controller.launch_sequence_activated?
end
```

Note that in this block, sender is a reference to the cancel_button widget, since it's the sender of the SEL_UPDATE message.

Making Choices with Radio Buttons

When you need your user to make a selection from a group of mutually exclusive options, you should consider using a group of FXRadioButton widgets. Radio buttons are an appropriate choice when the number of options is fixed and reasonably small. If the number of options is unknown until runtime, you're probably better off using a different user interface object such as a list or combo box that's designed to accommodate an arbitrary number of items. You'll also want to use a list-like widget if you're presenting the user with more than a few choices, because a long column of radio buttons is difficult to deal with visually.

Let's put together a short example program to demonstrate everything you need to know when working with radio buttons. First, although it's not required, it's good practice to use an FXGroupBox widget to visually group radio buttons. You can use either the FRAME_GROOVE or FRAME_RIDGE option with the group box to affect the style of the outline, and you can optionally assign a title to the group box:

Download radiobuttons1.rb

```
groupbox = FXGroupBox.new(self, "Options",
  :opts => GROUPBOX_NORMAL|FRAME_GROOVE|LAYOUT_FILL_X|LAYOUT_FILL_Y)
```

Figure 8.7: Use radio buttons for mutually exclusive choices.

Now you can add a couple of radio buttons to represent the different options:

Download radiobuttons1.rb

```
@radio1 = FXRadioButton.new(groupbox, "Good Enough")
@radio2 = FXRadioButton.new(groupbox, "Perfect")
```

While we're at it, let's add an instance variable to hold the index of the currently selected radio button and make sure that the first radio button is checked by default:

Download radiobuttons1.rb

```
@choice = 0
@radio1.checkState = true
```

Now, since there's still no link between the value of @choice and the radio buttons, let's connect each of the radio buttons to a block that will update @choice whenever one of the radio buttons is selected:

Download radiobuttons1.rb

```
@radio1.connect(SEL_COMMAND) { @choice = 0 }
@radio2.connect(SEL_COMMAND) { @choice = 1 }
```

Figure 8.7 shows what this example looks like running on Windows. If you run the program at this point and start selecting the two radio buttons, you'll observe a pretty serious problem right off the bat. It turns out that the group box doesn't do anything to enforce the mutual exclusivity of the radio buttons—it's just window dressing. To ensure that only one of the radio buttons in a group is selected at a time, we need to do something in our program to enforce that constraint.

A straightforward way to do this is to just connect each of the radio buttons in the group to a block that updates the state of the radio button based on the value of @choice:

Download radiobuttons2.rb

```
@radio1.connect(SEL_UPDATE) { @radio1.checkState = (@choice == 0) }
@radio2.connect(SEL_UPDATE) { @radio2.checkState = (@choice == 1) }
```

If you run the program again, you should find that it's behaving much better at this point. When you check the Perfect radio button, the Good Enough radio button is unchecked, and vice versa. Despite this success, we can imagine that this approach won't scale well for a larger number of options. The bookkeeping code required to manage the radio buttons' states would result in a lot of code clutter.

A more elegant way to handle this problem is to create an FXDataTarget instance to hold the selected choice number:

Download radiobuttons3.rb

```
@choice = FXDataTarget.new(0)
```

Now we can make this data target the target object for each of the radio buttons in the group:

Download radiobuttons3.rb

```
radio1 = FXRadioButton.new(groupbox, "Good Enough",
  :target => @choice, :selector => FXDataTarget::ID_OPTION)
radio2 = FXRadioButton.new(groupbox, "Perfect",
  :target => @choice, :selector => FXDataTarget::ID_OPTION+1)
```

With this change in place, we can get rid of all those calls to connect() for the radio buttons. Although this is clearly a lot cleaner than our previous attempt, there's a little snag. As things currently stand, we don't have any way of knowing when the value of @choice actually changes. That might be a significant problem if we want to change some other part of the GUI whenever a different choice is selected. Fortunately, an FXDataTarget is like any other FOX object in that we can connect it to a command handler:

Download radiobuttons3.rb

```
@choice.connect(SEL_COMMAND) do
  puts "The newly selected value is #{@choice.value}"
end
```

With this final change, we get all the benefits of using an FXDataTarget as the gatekeeper for any changes to @choice without losing any visibility we might want when those changes take place. Now we're going

to switch gears and take a look at a different technique for offering the user choices, and that's with the FXCheckButton widget.

Check Buttons: Yes? No? Maybe?

When you're dealing with an application setting that can take on one of two possible states, the FXCheckButton widget is probably your best choice to represent that setting.

Well, let me be more specific: it's probably your best choice when those two possible states are *opposites* of each other, like an "on" or "off" setting. Even though our radio button example from the previous section offered the user only two choices, a check button wouldn't have been an appropriate widget since Perfect is not the opposite of Good Enough.[2]

Check buttons can actually be in one of three possible states. In addition to the "checked" and "unchecked" states, a check button can be in an indeterminate, or "maybe," state. I've seen this feature used in different ways. For example, suppose you have a check button that you're using to indicate whether the selected articles in a news feed have been read. If they've all been read, the check button should be in the checked state, and if none of them have been read, the check button should be in the unchecked state. But what if some of the selected articles have been read and others haven't? In this situation, you might want to set the check button state to indeterminate. When a check button is in this state, it will appear to be checked but will also have a dimmed background (see Figure 8.8, on the next page). When the user clicks a check button in the indeterminate state, its state will change to unchecked, and at that point you're back to the basic checked/unchecked toggling. There's no way for the user to "click" a check button back into the indeterminate state.

The FXCheckButton class uses the values true, false, and MAYBE, respectively, to represent the checked, unchecked, and indeterminate states. You can set a check button's state via the checkState attribute and test its state using the checked?(), unchecked?(), and maybe?() queries:

```
checkbutton = FXCheckButton.new(...)
checkbutton.checkState = true
checkbutton.checked?   # returns true
checkbutton.unchecked? # returns false
checkbutton.maybe?     # returns false
```

2. They are merely enemies.

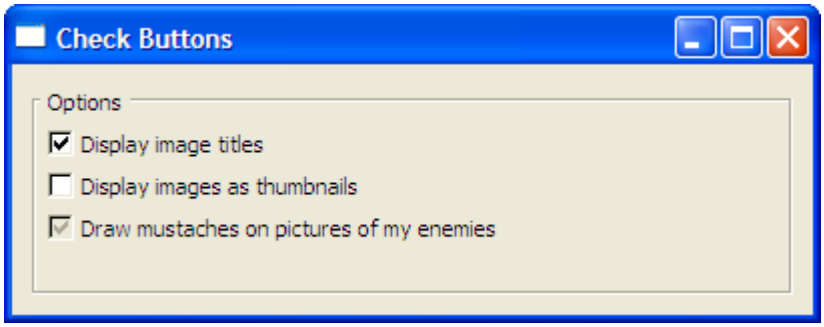

Figure 8.8: Check Buttons in the checked, unchecked, and maybe states

As was the case with the FXRadioButton, it's often convenient to connect a check button to a data target:

Download checkbutton.rb

```
@titles = FXDataTarget.new(true)
@thumbnails = FXDataTarget.new(false)
@mustaches = FXDataTarget.new(MAYBE)
groupbox = FXGroupBox.new(self, "Options",
  :opts => GROUPBOX_NORMAL|FRAME_GROOVE|LAYOUT_FILL)
titles_check = FXCheckButton.new(groupbox,
  "Display image titles",
  :target => @titles, :selector => FXDataTarget::ID_VALUE)
thumbnails_check = FXCheckButton.new(groupbox,
  "Display images as thumbnails",
  :target => @thumbnails, :selector => FXDataTarget::ID_VALUE)
mustaches_check = FXCheckButton.new(groupbox,
  "Draw mustaches on pictures of my enemies",
  :target => @mustaches, :selector => FXDataTarget::ID_VALUE)
@titles.connect(SEL_COMMAND) do
  puts "The new value for 'titles' is #{@titles.value}"
end
```

In this program, we construct a separate FXDataTarget object for each of the settings and then associate those data targets with the corresponding check buttons. Since @mustaches is initialized to the MAYBE value, the third check button (mustaches_check) will start in the indeterminate state. We can also connect each of the data targets to a block of code to be notified when their values change.

Figure 8.9: Use text fields to edit single lines of text.

The FXRadioButton and FXCheckButton widgets are all about letting the user make selections from a fixed set of choices, and they are some of the simplest tools that FXRuby gives you for providing that kind of functionality. Naturally, FXRuby provides other kinds of widgets that don't inherently limit the user's input to a fixed set of choices, and in the next section we'll take a look at one such widget, the FXTextField.

8.2 Editing String Data with Text Fields

The FXTextField widget is appropriate when you need to provide for the input and subsequent editing of single-line text strings. Figure 8.9 shows a couple of text fields from the example program we'll look at in this section. For working with multiline text, you should look at Chapter 10, *Editing Text with the Text Widget*, on page 135.

Most of the time, you're going to want to handle the SEL_COMMAND message from a text field. The FXTextField sends a SEL_COMMAND message to its target when the user presses the Return (or Enter) key after typing some text in a text field. It will also send a SEL_COMMAND message when the text field loses the keyboard focus (because the user clicked somewhere else or pressed the Tab key to shift the focus to some other widget).[3]

If you need a more fine-grained response to changes in the text field's contents, you should instead handle the SEL_CHANGED message. The FXTextField widget will send a SEL_CHANGED message to its target after every keystroke.

3. You can override this behavior by passing in the TEXTFIELD_ENTER_ONLY flag when you construct the FXTextField.

For some applications, you'll want to be able to limit the kinds of text that can be entered into a text field. The FXTextField supports a few different modes out of the box to deal with some of the more common cases. For example, when you're using a text field to accept entry of a password, it's common practice to mask the password text using asterisks. You can do this by passing in the TEXTFIELD_PASSWD flag when you construct the text field. Likewise, you can restrict the input in a text field to integer or floating-point values using the TEXTFIELD_INTEGER and TEXTFIELD_REAL modes.

If your application calls for some more complicated limits on the entered text, you can handle the SEL_VERIFY message that the text field sends to its target. This message is similar to SEL_CHANGED, but with an important difference: the SEL_VERIFY message is sent before the tentative changes are "committed," so to speak. For example, if we wanted to verify that the text begins with a letter and consists of only letters and numbers, we might do something like this:

Download textfield.rb

```
userid_text.connect(SEL_VERIFY) do |sender, sel, tentative|
  if tentative =~ /^[a-zA-Z][a-zA-Z0-9]*$/
    false
  else
    true
  end
end
```

Note that if the text *doesn't* match the expected pattern, the block returns true. This seems a little counterintuitive, but it's our way of telling FOX that the SEL_VERIFY message has been "handled" and that no further processing should be done. If the text matches and the block returns false, FOX will proceed and update the text field's contents.

Finally, as with the other widgets that we've talked about in this chapter, you can associate a text field with a data target:

Download textfield.rb

```
@name_target = FXDataTarget.new("Sophia")
name_text = FXTextField.new(p, 25,
  :target => @name_target, :selector => FXDataTarget::ID_VALUE)
@name_target.connect(SEL_COMMAND) do
  puts "The name is #{@name_target.value}"
end
```

If you don't require the SEL_VERIFY handling, this is the most convenient way to work with a text field. As shown, you can also connect the data

target to some downstream target object if you want to be notified of changes to the data target's value.

We are going to wrap up this chapter by looking at how you can incorporate help messages via tooltips and the status bar. Although these are both technically just another kind of display widget, not unlike labels, they're unusual in the sense that they always work in conjunction with other widgets to provide a kind of higher-level service for the application.

8.3 Providing Hints with Tooltips and the Status Bar

The tooltip is a special kind of pop-up window that knows to show itself whenever the mouse cursor rests in one particular spot for a few seconds. The tooltip asks the widget that the mouse cursor is pointing at what it's tooltip text is, and the tooltip displays that text. The tooltip will display this text for a short time, and then it will hide again; it will also hide as soon as you move the mouse cursor to a new location.

FXRuby makes it easy to add tooltips to your application. First, you need to create the FXToolTip object:

Download tooltipexample.rb

```
FXToolTip.new(app)
```

Note that there's only one tooltip object for the entire application, which may seem a little counterintuitive since you'll see the tooltip pop up all over the place!

Next, you need to specify the tooltip text that should be displayed for each widget, since the default tooltip text for a widget is empty.[4] For FXButton widgets and other widgets derived from FXButton, you can embed the tooltip text directly in the button label when you construct the button:

Download tooltipexample.rb

```
upload_button = FXButton.new(self, "Upload\tUpload Files")
```

Note that the tooltip text is separated from the button's label by a tab character.

4. You aren't required to specify tooltip text for widgets that don't need it. The tooltip is smart enough not to show itself when it has no text to display.

Many other widgets that don't have a label associated with them also allow you to set their tooltip text, using the tipText attribute:

Download tooltipexample.rb

```
dial = FXDial.new(self, :opts => DIAL_HORIZONTAL)
dial.range = 0..11
dial.tipText = "Volume"
```

Like the tooltip, the FXStatusBar widget is capable of displaying a context-sensitive help message about a widget when the mouse cursor hovers over that widget. Unlike the tooltip, the status bar is a permanent fixture on your application's main window—it doesn't just pop up briefly and then disappear again like a tooltip. Traditionally, the status bar is placed along the bottom edge of the main window and stretched to the full width of the main window, but that's not required if your application has some other layout needs.

For FXButton widgets and other widgets derived from FXButton, you can specify the help message for a widget directly in the button's label:

Download tooltipexample.rb

```
download_button = FXButton.new(self,
  "Download\tDownload Files\tStart Downloading Files in the Background")
```

Note that the status line help text is separated from the tooltip text by a second tab character. You can use the helpText attribute to specify the status line help text for widgets that don't have a label associated with them.

Download tooltipexample.rb

```
dial.helpText = "This one goes to eleven"
```

The widgets that we've looked at in this chapter are among the simplest widgets that FXRuby has to offer, and they primarily deal with setting and displaying single values. In the next chapter, we'll kick it up a notch and see what tools FXRuby provides for working with lists of values.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

FXRuby's Home Page
http://pragprog.com/titles/fxruby
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/fxruby.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com