1

Overview of Java UI Toolkits and SWT/JFace

This chapter outlines the three main Java user interface (UI) toolkits: AWT, Swing, and JFace. First I provide a brief introduction to all three, and then I compare them, highlighting some of the advantages SWT/JFace offers. SWT/JFace allows you to access native features easily, and programs based on SWT/JFace are considerably faster than those based on Swing in terms of execution speed. SWT/JFace is designed to be very flexible, so you can program using either the traditional approach or the model-view-controller approach. After reading this chapter, you should have a general overview of SWT/JFace. The chapters that follow introduce various aspects of SWT/JFace in detail.

Evolution of Java GUI Frameworks

This section covers the following Java graphical user interface (GUI) frameworks:

- **Abstract Window Toolkit (AWT):** The first and the simplest windowing framework.
- **Swing:** Built on AWT, Swing offers peerless components.
- **Standard Widget Toolkit (SWT) and JFace:** SWT is a native widget UI toolkit that provides a set of OS-independent APIs for widgets and graphics. JFace is a UI toolkit implementation using SWT to handle many common UI programming tasks.

This section outlines the evolution of the Java GUI framework and highlights the key features we'll compare and contrast in the next section.

Abstract Window Toolkit

The first version of Java, released by Sun Microsystems in 1995, enabled you to create programs on one platform and deliver the products to other Java-supported systems without worrying about the local environment — "Write Once, Run Anywhere." Most early Java programs were fancy animation applets running in Web browsers. The underlying windowing system supporting those applets was the Abstract Window Toolkit (AWT).

AWT has a very simple architecture. Components, graphics primitives, and events are simply perched on top of similar elements from the underlying native toolkit. A layer of *impedance matching* sits between the AWT and various underlying native toolkits (such as X11, Macintosh, and Microsoft Windows) to ensure the portability of AWT.

AWT 1.0 uses a callback delegation event model. Events are propagated or delegated from an event "source" to an event "listener." The interested objects may deal with the event, and the super-event handler is not required. The event model in AWT 1.1 was reimplemented from the callback delegation event model to an event subscription model. In AWT 1.1, the interested objects must register themselves with the components to receive notification on certain events. When the events are fired, event object are passed to registered event listeners.

AWT was slightly enhanced in later releases of Java. However, even the latest version of AWT fails to delivery a rich set of GUI components. Following is a list of components provided by AWT:

- `Button`
- `Canvas`
- `Checkbox`
- `Choice`
- `Container`
  - `Panel`
  - `ScrollPane`
  - `Window`
- `Label`
- `List`
- `Scrollbar`
- `TextComponent`
  - `TextArea`
  - `TextField`

To give you a more complete overview of the AWT user interface, I've created a simple GUI program. This tiny program allows the user to upload a photo to a server, or anywhere else. Figure 1-1 shows the user interface of the photo uploader implemented using Abstract Window Toolkit.

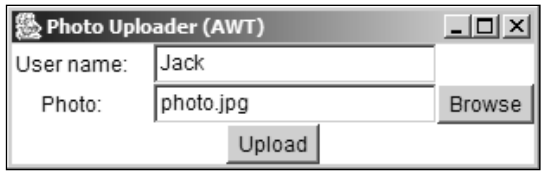

Figure 1-1

Click the Browse button to bring up the file selection dialog (see Figure 1-2). The name of the selected file is inserted into the text after the Photo label. The upload process starts when the user clicks the Upload button. The program exits when uploading is complete.

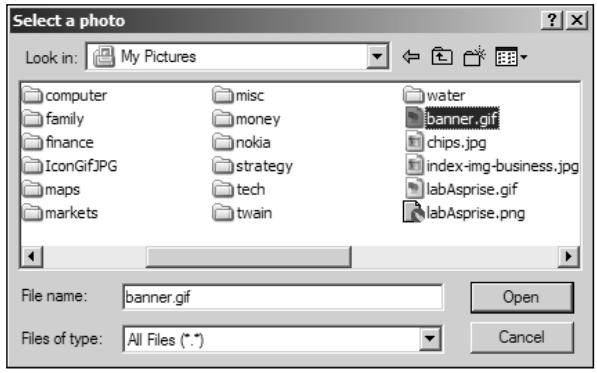

Figure 1-2

If you are familiar with Microsoft Windows systems, you may notice that the file selection dialog in Figure 1-2 is exactly the same as those used by native Windows programs. The Abstract Window Toolkit passes the call for file selection to the underlying native toolkit, i.e., Windows toolkit, and as a result, a native Windows file selection dialog pops up.

The Abstract Window Toolkit is sufficient for developing small user interfaces and decorations for Java applets, but it's not suitable for creating full-fledged user interfaces. Sun Microsystems recognized this as well and in 1997, JavaSoft announced Java Foundation Classes (JFC). JFCs consist of five major parts: AWT, Swing, Accessibility, Java 2D, and Drag and Drop. Swing helps developers to create full-scale Java user interfaces.

Swing

Swing is a pure Java UI toolkit built on top of the core Abstract Window Toolkit (AWT) libraries. However, the components available in Swing are significantly different from those in AWT in terms of underlying implementation. The high-level components in Swing are lightweight and peerless, i.e. they do not depend on native peers to render themselves. Most AWT components have their counterparts in Swing with the prefix "J." Swing has twice the number of components of AWT. Advanced components such as trees and tables are included. The event-handling mechanism of Swing is almost the same as that of AWT 1.1, although Swing defines many more events. Swing has been included in every version of Java since Java 1.2.

The main Swing packages are as follows:

- **javax.swing:** Contains core Swing components.
- **javax.swing.border:** Provides a set of class and interfaces for drawing various borders for Swing components.
- **javax.swing.event:** Contains event classes and corresponding event listeners for events fired by Swing components, in addition to those events in the java.awt.event package.
- **javax.swing.plaf:** Provides Swing's pluggable look-and-feel support.
- **javax.swing.table:** Provides classes and interfaces for dealing with JTable, which is Swing's tabular view for constructing user interfaces for tabular data structures.
- **javax.swing.text:** Provides classes and interfaces that deal with editable and noneditable text components, such as text fields and text areas. Some of the features provided by this package include selection, highlighting, editing, style, and key mapping.
- **javax.swing.tree:** Provides classes for dealing with JTree.
- **javax.swing.undo:** Provides support for undo and redo features.

In addition to the lightweight high-level components, Swing introduced many other features over AWT. Pluggable look-and-feel is one of the most exciting of the bunch. Swing can emulate several look-and-feels, and you can switch the look-and-feels at runtime. If you do not like any of them , you can even create your own. Other features include tooltip support, keyboard event binding, and additional debugging support.

The photo uploader program can be rewritten using Swing. Figure 1-3 shows the user interface of the Swing photo uploader with Windows look-and-feel; Figure 1-4 shows the user interface with Java metal look-and-feel.

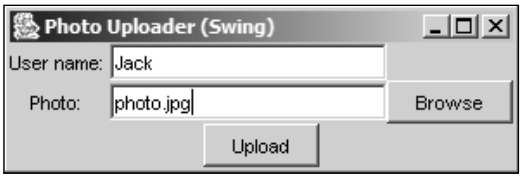

Figure 1-3

Figure 1-4

The Swing file selection dialog user interfaces for Windows look-and-feel and Metal look-and-feel are shown in Figures 1-5 and 1-6, respectively. The Swing file selection dialog with Windows look-and-feel looks similar to the AWT (i.e. the native dialog); however, they are quite different. Swing simply emulates

the Windows native file dialog. If you look carefully, you'll find that some features of Windows native file dialogs are missing in the Swing file dialog. In Windows native file dialogs, you can view the files using different modes: list, details, thumbnails, and so on. Additionally, more operations are available in the popup menu when you right-click. Both of these features are not available to Swing file selection dialogs.

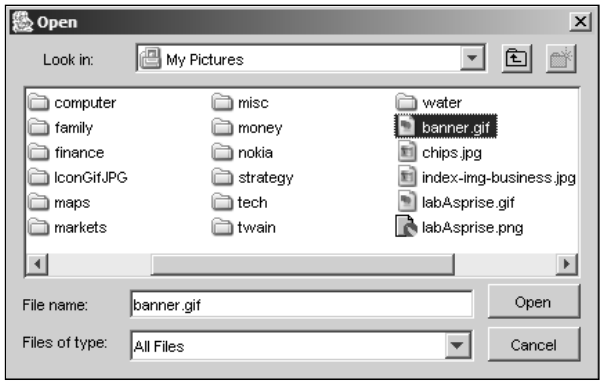

Figure 1-5

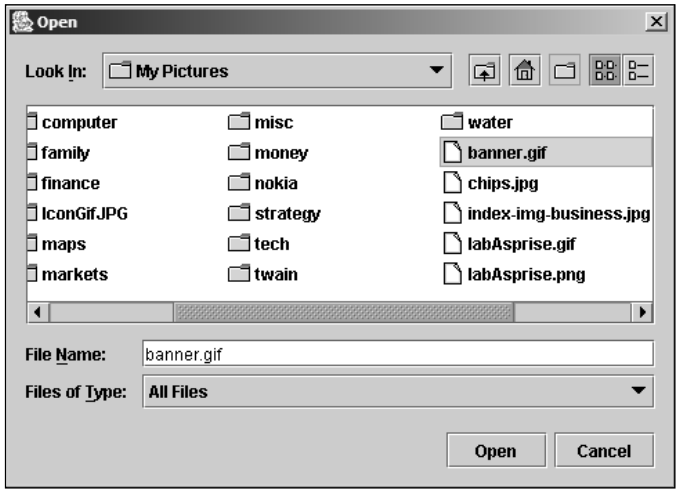

Figure 1-6

Swing fails to support native features of the underlying system. Another obstacle to widespread usage is that programming with Swing is very complex.

Swing is so powerful that you can use it to create full-scale enterprise Java user interface programs. So why do we see so few Swing-based GUI programs? James Gosling, creator of the Java language, said during a keynote presentation at a Mac OS X conference that there is a "perception that Java is dead on the desktop." Complexity of building Swing GUIs, lack of native features, and slow running speed are some of obstacles keeping Swing from succeeding on desktops.

Is any other Java GUI toolkit available that can create full-featured user interface programs? The answer is yes. Standard Widget Toolkit (SWT), along with JFace, provides a complete toolkit for developing portable native user interfaces easily.

SWT and JFace

Eclipse is an open source universal tool platform dedicated to providing a robust, full-featured, industry platform for the development of highly integrated tools. IBM, Object Technology International (OTI), and several other companies launched the Eclipse project in 2001. Today, the Eclipse Board of Stewards includes companies such as Borland, Fujitsu, HP, Hitachi, IBM, Oracle, Red Hat, SAP, and Sybase. With more than 3 million downloads, Eclipse has attracted a huge number of developers in over 100 countries.

The Eclipse platform defines a set of frameworks and common services that are required by most tool builders as common facilities. One of the most important common facilities is the portable native widget user interface. The Standard Widget Toolkit (SWT) provides portable native user interface support, as well as a set of OS-independent APIs for widgets and graphics.

Built on SWT, JFace is a pure Java UI framework handling many common UI programming tasks. The following subsections introduce SWT and JFace in detail.

Figure 1-7 shows the Eclipse platform's native user interface — in this case, Windows. SWT is integrated tightly with the underlying native window system.

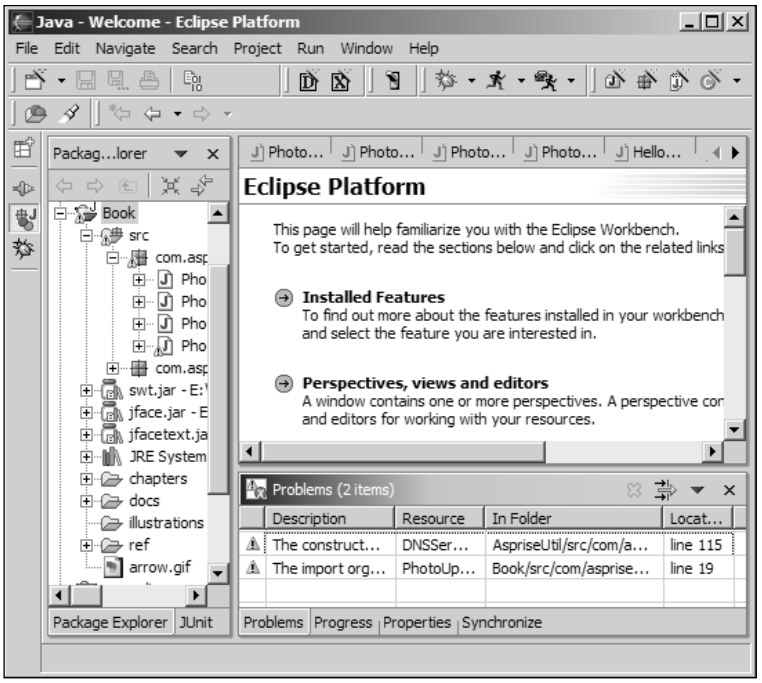

Figure 1-7

Standard Widget Toolkit

SWT is analogous to AWT and Swing in Java except that SWT uses a rich set of native widgets. AWT widgets are implemented directly with native widgets, so to be portable it has to take the least common denominator of all kinds of window systems. For example, while Windows supports a tree widget, Motif does not. As a result, AWT cannot have the tree widget. Swing tackles this problem by emulating almost all kinds of widgets. However, this emulation strategy has some serious drawbacks. First, the emulated widgets lag behind the look and feel of the native widgets, and user interaction with the emulated widgets is quite different. Second, although Swing has been improved, Swing user interfaces are still sluggish.

SWT employs a slightly different strategy. It defines a set of common APIs available across supported window systems. For each native window system, the SWT implementation utilizes native widgets wherever possible. If no native widget is available, the SWT implementation emulates it. As mentioned previously, Windows has a tree widget so SWT simply uses the native tree widget on Windows systems. For Motif, because it does not have a tree widget, the SWT implementation provides a proper emulated tree widget. In this way, SWT maintains a consistent programming model on all platforms and takes full advantage of any underlying native window systems.

The user interface of the photo uploader, rewritten using SWT, and the file selection dialog are shown in Figures 1-8 and 1-9, respectively.

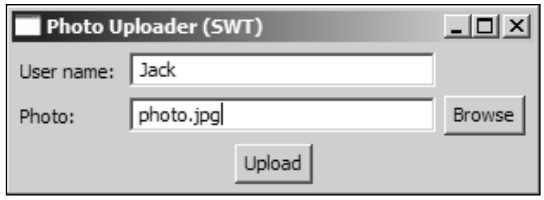

Figure 1-8

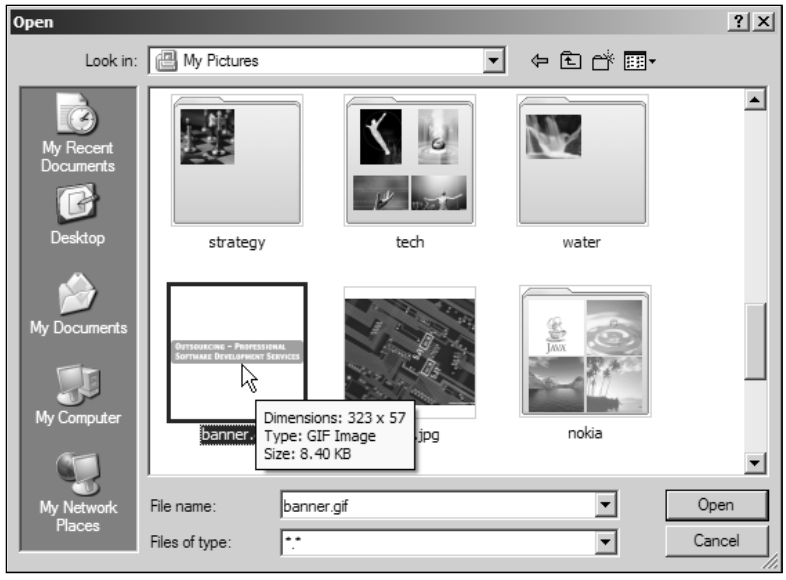

Figure 1-9

SWT is tightly integrated with the underlying native window system. Chapter 2 discusses the implementation of SWT and its advantages. Although SWT does not support pluggable look-and-feel (who needs Windows or Metal look-and-feels on a Mac, anyway?), it provides a number of other invaluable features: native UI interactions (such as drag and drop) and access to OS-specific components (such as Windows ActiveX controls like Microsoft Word, Acrobat Reader, and so on).

SWT enables developers to create native user interfaces with Java. However, most programmers with experience developing user interfaces on Windows, Linux, or any of the other platforms, know that developing a GUI is a very complicated and time-consuming process. Creating a native user interface with Java is no exception. Fortunately, Eclipse provides a UI toolkit named JFace to simplify the native user interface programming process.

JFace

JFace is a UI toolkit implemented using SWT to handle many common user interface programming tasks. It is window system–independent in bots, its APIs, and implementation. JFace is designed to work with SWT without hiding it (see Figure 1-10).

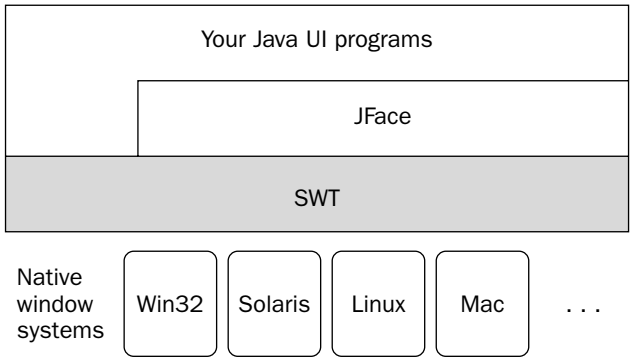

Figure 1-10

JFace offers the following components:

- **Image and font registries:** The image and font registries help the developer to manage OS resources.
- **Dialogs and wizards**
- **Progress reporting for long-running operations**
- **Action mechanism:** The action mechanism separates the user commands from their exact whereabouts in the user interface. An action represents a user command that can be executed by the user via buttons, menu items, or toolbar items. Each action defines its own essential UI properties, such as label, icon, tooltip, and so on, which can be used to construct appropriate widgets to present the action.
- **Viewers and editors:** Viewers and editors are model-based adapters for some SWT widgets. Common behaviors and high-level semantics are provided for those SWT widgets.

SWT/JFace Advantages

Compared with AWT and Swing, SWT/JFace offers many advantages, as described in the subsections that follow.

Full Support for Native Features

SWT/JFace is tightly integrated with the underlying native window system. For example, you can create Windows user interface programs with SWT on Windows, and they look and behave the same as those developed using Visual C++. Native features are available to SWT. This is a great advantage for the user.

Let's take the photo uploader as an example. When the user hits the Browse button, a file selection dialog pops up for the user to select the photo to be uploaded. In most of the cases, the user has quite a number of pictures. File names may help the user to identify the proper photo. However, file names are not intuitive enough to the user, especially if these photos are just exported from a digital camera. The best way to assist the user in choosing the proper photo is to provide a thumbnail preview.

The file chooser in Swing does not provide the picture preview function. Sun's Swing tutorial offers a way to do this: extending the file chooser with an accessory component to display a thumbnail of the selected file. Figure 1-11 shows a custom file chooser implemented using Swing.

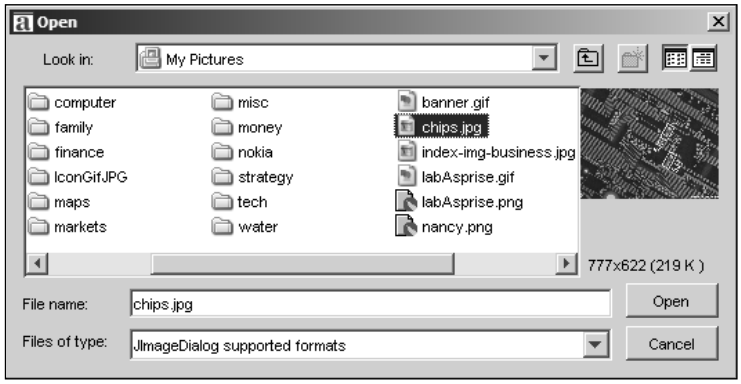

Figure 1-11

The Windows file chooser (refer to Figure 1-9) invoked from SWT offers more features than the custom Swing file chooser. In the custom Swing file chooser, the user has to click each file to view its corresponding thumbnail. However, the Windows file chooser displays all the thumbnails to the user (even photos in subfolders!) so he or she can easily find the proper file. Behind the scenes, Swing cannot display bitmap files (BMP) even though they're so popular on Windows and OS/2. On the other hand, SWT/JFace handles almost any kind of native image formats very well.

While the custom Swing file chooser may ultimately look even better than the Windows file chooser, it still looks strange to the user. The screen fonts in Figures 1-8 were smoothed with clear type method, except those from Swing. This means that the user selected clear type method to smooth the screen fonts' edges using the display panel of the native OS; however, Swing is unable to access this preference setting.

Such a small defect may not seem important to the developer, but to the user it is frustrating — especially when these small defects start to add up. Naturally, this leads to the loss of valuable customers. You can always hear such stories from the sales or marketing guys. This could be one of most important reasons why "Java is dead on the desktop."

Many components in Swing — including the file chooser — need improvement. You could fix them one by one and create a complete toolkit, but it is not necessary. With SWT/JFace, you do not have to create your own UI toolkit and you can still have total control of native features. If you are not satisfied with any aspect of SWT/JFace, you can always modify the source code provided.

Speed

Most UI programs using SWT are more responsive than those using Swing. The following table provides an informal comparison of speeds of Swing and SWT.

	Swing	**SWT**
Time used from the click of the Browse button to the display of a file selection dialog	a. 2.39s b. 2.53s c. 2.46s Avg. 2.46s	a. 0.63s b. 0.58s c. 0.68s Avg. 0.63s
Time used from the display of the selection dialog to the dialog fully loaded	a. 1.48s b. 1.40s c. 1.44s Avg. 1.44s	[Fully loaded immediately. Approximate 0 sec.]
Total time used	3.90s	0.63s

Setup of the experiment: The photo uploader implemented with Swing was executed three times (a, b, c). For each run, the time was documented using a stopwatch from the click of the Browse button to the display of a file selection dialog and from the display of the selection dialog to the dialog fully loading. The average total time taken is computed. This process is repeated for photo uploader implemented with SWT.

Testing environment: Windows XP Professional; Sun Java 1.4.1 JRE; Mobile Intel Pentium 1.7 GHz CPU, 512MB memory.

The results from the experiment may not be entirely accurate, but we can make some generalizations. SWT runs significantly faster than Swing. It takes Swing up to six times as long to fully load the file selection dialog.

GUI design is an art as well as a science, so this comparison tells only part of the story. Trying out SWT and Swing applications reveals another part of the story: SWT is clearly superior to Swing in terms of visual perception.

SWT is designed to be very efficient. Unlike AWT (which uses a separate peer layer), SWT is a thin layer on top of the native window systems. To further reduce the overhead and potential incompatibilities, SWT attempts to avoid sugar-coating the limitations of the underlying window system. For example, SWT does not attempt to hide the existence of limitations on cross-threaded access to widgets.

A sluggish user interface has always been one of the top complaints about Swing. One of the primary problems was that Swing did not leverage much hardware acceleration. Over the past few years, this has improved, and a certain level of graphics acceleration was added with JDK 1.4. While it may be technically faster with this release, the perceived speed is still disappointing (especially to anyone not using high-end workstations).

Our experiment also shows that Swing is still slow compared with SWT. If a product has to satisfy the diverse execution environments (machines with different CUP power capabilities), SWT is the most suitable toolkit candidate.

Portability

SWT provides a set of common programming APIs that developers can use to create portable applications on all of the SWT-supported operating systems. Following is a list of SWT (v2.1)–supported OSs:

- Windows 98/ME/2000/XP/CE
- Linux (x86/Motif; x86/GTK2)
- Solaris 8 (SPARC/Motif)
- QNX (x86/Photon)
- AIX (PPC/Motif)
- HP-UX (HP9000/Motif)
- Mac OS X (Mac/Carbon)

Easy Programming

Some people bad-mouth SWT because the developer needs to take care of garbage collection on operating system resources, rather than the UI toolkit itself. It is true that SWT requires developers to track and dispose of these resources, but programming these kinds of tasks — and programming in SWT/JFace in general — is very straightforward. You could get started with SWT/JFace very quickly, although programming experience on a native user interface is an advantage.

The two simple rules developers should follow to develop UI programs with SWT are as follows:

- If you created it, you must dispose of it.
- Disposing of the parent disposes of the children (labels, text fields, and buttons).

Chapter 2 covers the resource management topic in detail. Here, I use code snippets to show you how to apply the preceding rules. These rules are very easy to adhere to, using the following steps:

1. In the photo uploader program, create a shell (window) first:

```
Display display = new Display();
Shell shell = new Shell(display);
```

2. Next, create labels, text fields, and buttons with the shell as their parent:

```
Label labelUser = new Label(shell, SWT.NULL);
Label labelPhoto = new Label(shell, SWT.NULL);
Text textUser = new Text(shell, SWT.SINGLE | SWT.BORDER);
Text textPhoto = new Text(shell, SWT.SINGLE | SWT.BORDER);
Button buttonBrowsePhoto = new Button(shell, SWT.PUSH);
Button buttonUpload = new Button(shell, SWT.PUSH);
```

3. Click the Upload button and execute the following code:

```
uploadPhoto(textUser.getText(), textPhoto.getText());
shell.dispose();
```

This disposes of the shell. As I said in the preceding text (the first rule), the shell is created and it must be disposed of. This rule does not apply to labels, text fields, and buttons; they are created, but they are never disposed of explicitly. However, when the shell is disposed of (the second rule in the preceding text) all of its children are disposed of, too.

Some may complain that the shell still has to be disposed of explicitly, but this process is necessary with almost all UI toolkits — including Swing. Chapter 2 covers programming in greater detail.

Flexibility

SWT/JFace is designed to be very flexible. With SWT/JFace, you can use either of the following methods for programming:

- **The traditional approach:** The application model data must be transformed and copied from corresponding data structures to native UI components.
- **The model-view-controller (MVC) approach:** The MVC pattern is a classic design pattern. It separates the application object (model) from the way it is represented to the user (view) and from the way in which the user controls it (controller). Most MVC classes can be found in JFace.

Both approaches have advantages and disadvantages. The traditional approach is simple and easy to learn. The MVC approach requires much more time for developers to master, but it is more maintainable and extensible. In Swing, only the MVC approach is allowed.

Having both approaches available in SWT/JFace shortens the learning curve. Programming in the traditional approach is easy to learn, and the basic knowledge acquired from the traditional approach helps the developer understand the mechanisms behind the MVC approach.

Figure 1-12 shows a sample application displaying nesting of categories.

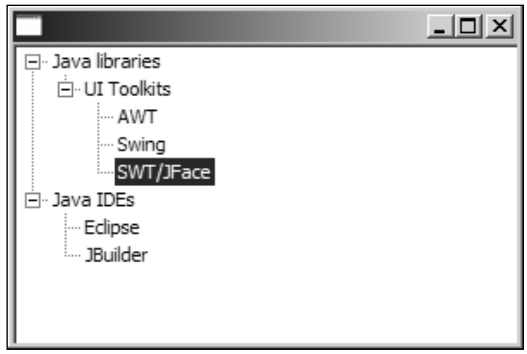

Figure 1-12

Creating the Application Model Data

The application data can be modeled as following:

```
/**
* Represents a category of items.
*
*/
class Category {
    private String name;
    private Vector subCategories;
    private Category parent;

    public Category(String name, Category parent) {
        this.name = name;
        this.parent = parent;
        if(parent != null)
            parent.addSubCategory(this);
    }

    public Vector getSubCategories() {
        return subCategories;
    }

    private void addSubCategory(Category subcategory) {
        if(subCategories == null)
            subCategories = new Vector();
        if(! subCategories.contains(subcategory))
            subCategories.add(subcategory);
    }

    public String getName() {
        return name;
    }

    public Category getParent() {
        return parent;
    }
}
```

The category represents a category of items. It may have subcategories. Category data can then be constructed as follows:

```
Vector categories = new Vector();

Category category = new Category("Java libraries", null);
categories.add(category);

category = new Category("UI Toolkits", category);
new Category("AWT", category);
new Category("Swing", category);
new Category("SWT/JFace", category);

category = new Category("Java IDEs", null);
categories.add(category);

new Category("Eclipse", category);
new Category("JBuilder", category);
```

A vector named "categories" contains a list of top-level categories (categories that have no parent category).

The application model data is ready. The category display function, using both the traditional and MVC approaches, is presented in the subsection that follows.

Building the Tree Up with the Traditional Approach

The following code uses the traditional method:

```
final Tree tree = new Tree(shell, SWT.BORDER);

/**
 * Builds up the tree with traditional approach.
 *
 */
public void traditional() {
    for(int i=0; categories != null && i < categories.size(); i++) {
        Category category = (Category)categories.elementAt(i);
        addCategory(null, category);
    }
}

/**
 * Adds a category to the tree (recursively).
 * @param parentItem
 * @param category
 */
private void addCategory(TreeItem parentItem, Category category) {
    TreeItem item = null;
    if(parentItem == null)
        item = new TreeItem(tree, SWT.NONE);
    else
        item = new TreeItem(parentItem, SWT.NONE);
```

```
        item.setText(category.getName());

        Vector subs = category.getSubCategories();
        for(int i=0; subs != null && i < subs.size(); i++)
            addCategory(item, (Category)subs.elementAt(i));
    }
```

The implementation of the traditional method is straightforward. For each of the top-level categories, a tree item is created directly under the root of the tree. For other categories, tree items are created under tree items representing their parent categories.

Building the Tree Up with the MVC Approach

Following is the code for building the tree up using the MVC approach:

```
    final Tree tree = new Tree(shell, SWT.BORDER);

    /**
     * Builds up the tree with the MVC approach.
     *
     */
    public void MVC() {

        TreeViewer treeViewer = new TreeViewer(tree);

        treeViewer.setContentProvider(new ITreeContentProvider() {
            public Object[] getChildren(Object parentElement) {
                Vector subcats = ((Category)parentElement).getSubCategories();
                return subcats == null ? new Object[0] : subcats.toArray();
            }

            public Object getParent(Object element) {
                return ((Category)element).getParent();
            }

            public boolean hasChildren(Object element) {
                return ((Category)element).getSubCategories() != null;
            }

            public Object[] getElements(Object inputElement) {
                if(inputElement != null && inputElement instanceof Vector) {
                    return ((Vector)inputElement).toArray();
                }
                return new Object[0];
            }

            public void dispose() {
                //
            }

            public void inputChanged(Viewer viewer, Object oldInput,
                                                    Object newInput) {
                //
```

```
            }
        });

        treeViewer.setLabelProvider(new LabelProvider() {
            public String getText(Object element) {
                return ((Category)element).getName();
            }
        });

        treeViewer.setInput(categories);

    }
```

The MVC code is a little tedious, especially those two inner classes. Basically, the registered content provider `ITreeContentProvider` starts to provide content to the tree when the `setInput` method is called. Because the content provided comprises raw objects, the tree needs to consult the label provider `LabelProvider` about the text and image representation of those objects.

In this simple application, the traditional approach seems to be the preferred way. It is very straightforward. However, the MVC approach will shine if the application needs to be extended to support more features, such as category sorting.

If the requirement is not particularly complex and future changes are not likely to be big, then the traditional approach provides a fast path to your mission. On the other hand, if you are developing a large, complex system, the MVC approach could help you to create more scalable and maintainable software.

More on MVC is presented in Chapter 2.

Maturity

Ever since the first version of SWT was released as part of the Eclipse platform in 2001, companies have built their commercial software with SWT/JFace as the UI toolkit. These companies include:

- C++/Fortran compilers, Intel
- WebSphere Application Studio, IBM
- XDE Professional, Rational Software (now under IBM)
- ColdFusion MX, Macromedia
- UML modeling tool, Embarcadero

Summary

This chapter surveyed several major Java UI toolkits. Built on the Abstract Window Toolkit (AWT), Swing is a powerful UI toolkit with peerless components. Standard Widget Toolkit is a portable native UI toolkit, and the JFace framework simplifies programming with SWT by handling many common UI programming tasks. Compared to AWT and Swing, SWT/JFace offers many advantages such as full support of native features, fast execution speed, and flexibility of programming styles.