CHAPTER 9

Interface Design Tools

Graphical user interfaces are inherently visual in their appeal and in their use. The way the interface looks has a lot to do with how well users can find what they need and effectively accomplish their goals. A user interface frequently conveys the personality of its user base or the company that created it. A user interface might be conservative, spartan, frilly, soft, aggressive, hip, cool, or efficient. Very few programmers have the skills or the interest to make these differentiations in the interface. The visual organization of the interface has much less to do with model-view-controller and everything to do with the culture and habits of the intended user community. Programmers rarely have the time or inclination to become involved enough in the user community to make such design decisions.

Up to this point, you have built up a set of tools and techniques for programmers to create user interfaces. This chapter addresses the tools that empower designers and artists. These professionals possess significant and important skills, but programming is generally not among them. Interface design tools (IDT) have been developed to support the visual skills of other professionals in the design process. Some use interface design environments to describe such tools, but the term IDE has been co-opted for integrated design environments, which are interactive tools that support the entire program development process. This chapter is only concerned with those parts that design the graphical user interface.

Though the goal is to support artists and designers as part of the user-interface development process, most of this chapter is concerned with the software architecture to create such tools. This architecture is complicated by the fact that the interface design tool is a program that is written and supported by a completely different group of people than those who are creating user interfaces. Usually they are in completely different companies and their software is mutually opaque. The IDT developers cannot
know about all of the possible user interfaces to be built and all of the new widgets that will be created for use in those interfaces. On the other hand, the applications developers cannot be expected to look into or modify the interface design tools. Most IDTs are built around a plug-in architecture, which is a dynamic variant of the abstract model architecture discussed in Chapter 7.

In addition to the abstract model interfaces, you need reflection or introspection. Reflection is the ability at run time to find out information about the program currently running. Smalltalk and other interpreted languages had good reflection capabilities. Until Java and C# were developed, most compiled programming languages did not have good reflection capabilities and, thus, several mechanisms were developed to allow the programmers to provide the run-time information. However, programming languages with good reflection capabilities allow for a much simpler programming model than those that do not. Microsoft attempted to provide reflection-like capabilities with their COM and ActiveX libraries. These approaches are much more cumbersome than language-based reflection. Most of this chapter assumes the presence of reflection. Where appropriate, the historical alternatives are briefly discussed.

The key reflection features needed are (1) the ability to create an object of some class whose name is not known until run time, (2) the ability to explore the set of methods defined on any object of any class, and (3) the ability to select and invoke a method whose identity is not known until run time. Optionally, it is helpful if programmers can add information to their classes and methods that can be retrieved through the reflection mechanism. This is known as annotation.

The important issues in creating an interface design tool are creating the user interface visual layout, managing the properties associated with widgets, and, finally, establishing the binding between the user interface design and the code that implements the interface.

**LAYOUT DESIGN**

The most prominent feature of an IDT is the interactive design of user interface layouts. The nature of the IDT is strongly influenced by the layout-management mechanism of the underlying widget toolkit (see Chapter 9). The first published IDT was Buxton's MenuLay, similar to Figure 9.1. The system had a means for creating sketches of objects drawn with a pen and assembling these sketches into a palette, as shown across the bottom of Figure 9.1. The user could then select sketches from the palette, drag them onto the user interface, and attach callback names to them. Although the sketching tools were crude and the binding of events to callbacks was very simplistic, this tool formed the inspiration for Apple HyperCard and later Microsoft Visual Basic. In many ways, layout design tools have not changed in 20 years. The paradigm is still the selection of objects and placing them in the design.

The overwhelming advantage is that it is far easier to drag an object into a visually pleasing position than it is to change integer numbers in source code, recompile, and then view the result.

![Figure 9.1 - Early screen layout tool](image)

Almost immediately, the simple "place it here" style was replaced by drawing a rectangle into which the object should be placed. This provides the fixed-position layout design from Chapter 5. Fixed-position layouts were great when screens were small and there was generally only one window open on the screen at a time.

The initial problem is to incorporate a layout model with the interactive design metaphor. This section briefly discusses how edge-anchored layouts and variable intrinsic size work with interactive tools. The next problem is to work with live widgets in the design tool. The inputs for design are different from the inputs for interaction and the widget model must deal with this issue. The last problem of a layout tool is finding and presenting a set of widgets that are not known at the time the IDT is implemented. This is key to the extensibility of IDT tools.

**Edge-Anchored Layouts**

With the advent of many windows with variable size, fixed layout was just not adequate. The MIKE system introduced dynamic coordinates. These are essentially the edge-anchored layout mechanism found in X and modern Visual Basic. C# uses the layout design tool to specify a rectangle for a widget's placement and then uses the anchor property editor shown in Figure 9.2 to establish the connection to the edges. A similar mechanism was used in NeXTStep and has continued into Mac OS X.
MIKE took a different approach to the edge-anchored user interface. MIKE divided the screen into nine regions, as shown in Figure 9.3. When the user draws out the rectangle for a widget, its anchor properties are inferred from where the rectangle edges fall. For example, widget A would be of fixed width with both edges anchored to the left edge of the window. Widget A has its top anchored to the top and its bottom anchored to the bottom so that it will grow vertically with the window. Widget B has its edges anchored a fixed distance from the windows edges, it will grow and shrink with the window. Widget C will have a fixed vertical size anchored to the bottom of the window. Its left edge is anchored, but its right edge will stay about 70% of the distance between the window’s left and right edges. The key idea is that anchors occur near the edges and can be directly inferred from where the widget is drawn. The combination of the MIKE and C# approaches would be a very effective layout design tool.

A second difficulty is in visually interacting with the widget tree. Intrinsic size layouts are very tree oriented. Layouts are produced by combining various primitive layout schemes to produce the desired result. However, the structure of the tree is invisible and ambiguous on the screen. Figure 9.4 shows a fragment of a Microsoft Word window. If this had been created using variable intrinsic size, its tree structure would be that shown in Figure 9.5. The problem lies in the fact that the “Word window”, “Menus and toolbars”, “Menu bar” and “File” menu headers all have the same geometric position for their top edge. The left edge of “Menus and toolbars” is completely occupied by the left edges of “Menu bar”, “Toolbar” and “Toolbar 2”. There is no unambiguous way to select its left edge. Because of the way that nested widgets are used to create layouts, a container might frequently have all of its space filled by its child widgets, leaving no point that can be used to select the parent widget. There are various ways around this problem, but they all break the smooth feeling of interactively laying out widgets. Many interactive tools resolve this problem by adding new, special-purpose layout managers that mimic fixed-position and edge-anchored layout models.
widgets never seen by the implementers of the design tool. This means that live
widget implementation must be used by the design tool so that the user will see how
the interface will actually appear. There is a problem when trying to design with live
widgets. When the user places the mouse on the delete radio button to drag it to a
new location, the live widget will receive the mouse-down event and begin selecting
itself to be set. This is not what you want. What you want is to drag the widget.
The solution to this problem is found in the top-down event-handling strategy
from Chapter 3. In the top-down strategy, the event is first passed to the interface
design tool, which can choose whether to pass the event on to its child widget. In the
case of paint or resize events, the design tool passes them on to the child widgets so
that they will appear as they should. With mouse events, however, the design tool
retains those events for its own user-interface needs. The child widgets being moved
around the screen receive instructions to change their bounds but never any mouse
events. Attention to the event-handling strategy of the widget system takes care of
the problem.

Figure 9.6 – Laying out live widgets

Finding All of the Widgets

Suppose you want your interface design tool to handle new widgets that the tool
builders have never seen and use them as easily as the original widget set. Chapter 7
showed how an IDT might be implemented using an abstract drawing widget. This
approach, however, requires that the model know about all object classes at compile
time. This is not acceptable for an extensible IDT.

Using reflection, you can replace the model’s list of classes with a text file that
contains a list of class names for subclasses of Widget. The reflection capabilities of
Java or C# allow for new instances of these widgets to be dynamically created from
their names. Because all widgets have a bounds() and a repaint() method, the IDT
has all of the access it needs to position and display any widget. Figure 9.7 shows sam-
ple Java code for retrieving a widget given a class name.
public class WidgetInfo : Attribute
{
    public String menuName;
    public String menuIconFileName;
    public WidgetInfo(string mName, String mIconFile)
    {
        menuName = mName;
        menuIconFileName = mIconFile;
    }
}

[ WidgetInfo("My Special Widget", "Images\special.png") ]
public class MySpecialWidget : Widget
{
    ...

    public String widgetMenuName(Widget aWidget)
    {
        Attribute [] attributes = Attribute.GetCustomAttributes(aWidget.GetType());
        foreach (Attribute attr in attributes)
        {
            if (attr is WidgetInfo)
            {
                return ((WidgetInfo)attr).menuName;
            }
        }
        return aWidget.GetFullTypeName();
    }
}

Figure 9.9 – Annotating widgets in C#
PROPERTIES

There is much more to designing an interface than positioning widgets in a layout. A simple button has foreground color, background color, text color, font sizes and styles, border styles, border widths, the text to be displayed in the button, any icon on the button, text for a ToolTip to explain the button, whether there should be a different icon displayed when the button is pressed, and a variety of other pieces of information. These are all called properties and they are a key part of the interface design. When designing a widget, you should create as many of the design decisions into properties rather than in special code. Setting of properties is essentially a process of making choices rather than writing code.

As with layout interaction, the property setting interface must be able to deal with the properties of widgets that were not known when the IDT was created. The ability for the IDT to discover newly implemented properties is an important. This is further complicated by the fact that new properties might have new data types. Specifying such property values also complicates the IDT architecture. When a design is complete and all of the properties are set, there must be some means for saving that design in the IDT and then loading it at run time when the application is executed. This is primarily a property-saving problem. Lastly, the properties allow an interface design to be localized to a variety of cultures and languages. This is discussed more extensively in Chapter 10, but its implementation foundation is discussed here.

Figure 9.10 shows the Jigloo\textsuperscript{3} interface design tool for Java/Swing. A radio button is selected and some of its properties are shown at the bottom of the window. The foreground property is selected and a color editor has been opened to allow the user to select a color by looking at choices rather than typing RGB values. A variety of kinds of properties are available, including Booleans, fonts, colors, lists of choices, icon filenames, and integer numbers. An IDT must be able to adapt to whatever properties a widget might have and allow the user to edit those properties.

Access to properties was introduced in Chapter 4. Some languages like C\# provide field accessors that allow properties to be defined as fields with two hidden get and set methods to control access to that field. Java has no such mechanism, so it uses the following pattern method pairs:

\begin{verbatim}
 type getPropertyName();
 void setPropertyName(type newValue);
\end{verbatim}

Either of these two mechanisms can be handled using reflection. The Java Class class has a method called getMethods() that returns an array of Method objects. Method objects reveal the name, return type, and argument types of the method. It is quite easy to write the code necessary to detect property method pairs. Using the Method objects, you can also write code to invoke the methods to get their property values and to change their property values. Getting property names and their values, you can construct the property list shown in Figure 9.10. The property values can be displayed using the toString() method defined on all Java objects. A similar technique is possible in C\# by using a type object to retrieve all public fields of a Widget. The property list is then built in a similar way.

With this approach to editing properties, four problems arise. The first problem is parsing property values typed in by a user. The second problem occurs when the property names are defined for the consumption of programmers rather than designers. The third problem occurs when a property is intended for software use, but not to be exposed to the designer interface. The final problem is the use of the special editors such as the color editor on the foreground property.

Systems like ActiveX and Java Beans resolve these problems by creating special libraries that associate descriptive information with widget definitions. Java Beans provides methods that use reflection to access properties, but allows the programmer to add other information.

A better solution is to use the annotation facility. For a method or field to be included as a widget property, you can require a WIDGET\_PROPERTY annotation that includes the public name of the property. This resolves both the naming and the interactive access problems. A String\_Param annotation can name a method that will translate a string into an appropriate property value. A Property\_Editor annotation can name the class of a widget that can edit a property's value and produce a new value.

This approach should again be contrasted with tools like Flash and Dreamweaver that began as media editing tools rather than user-interface design tools. Dreamweaver, for example, is closely tied to the HTML standard, which has no concept of abstraction or widget development. The HTML standard defines the set of objects that can appear in a document and the set of valid properties that those objects can have. Flash is not subject to an external standards body, but it has similar characteristics. There are a fixed number of graphical object types and they have a fixed number of properties each with a clearly defined type. Unlike the general IDT, these tools have complete knowledge of the properties that they must manage. The property set is fixed at tool creation time.
Storing Resources

The preceding section discussed how tools can be built that know about widgets and can provide a designer with an interactive way to create a new user interface. After a user interface has been designed, the IDT needs a way to save the design and make the design available as part of a deployed application. An interface design tool has a model that it is editing. This model contains information about how the user interface is to be structured. This is different from the model that the user interface is to manipulate. Figure 9.11 shows a user-interface design in progress. The model for the application being designed is shown in Figure 9.12. The model that the IDT is working on is shown in Figure 9.13. The application will be editing information about dogs while the IDT is editing information about widgets.

![Figure 9.11 - User interface being designed](image)

```java
public class Dog {
    int heightInches;
    int weightInPounds;
    int ageInMonths;
}
```

![Figure 9.12 - Dog model](image)

As with any other interactive application, the IDT must have a means for saving its model. This is complicated by the fact that the interface design must be connected to the application that implements the user interface. This can be done in several ways. One of the earliest mechanisms for resource storage used text files consisting of property names and property values. Generally, there was one line per property. This gets a little complicated when there are many designs that make up an application's user interface. The X Toolkit* designed for X Windows® created a hierarchical property naming system to resolve the problem. When a user interface is initialized, the property files are read to provide the settings for the widgets.

A problem with this approach is that the property resource files get separated from the code files. The problem is not a design or development-time separation, which is a good thing. The problem is in what is delivered as an application. If a deployed application is a single file with all of its resources and code bundled together, it is easier to manage, install, and support. If the resources and the code are in separate files, they can get inadvertently detached from each other. This makes application installations more easily damaged and requires more support expertise. Without the resources created by the IDT, the application has no information about how to structure the interface. The earlier Apple Macintosh solution avoided this resource-code separation by creating special modifications to the file system. Every file on a Mac had a data fork and a resource fork. The data fork is the traditional stream of bytes that you expect from a file. The resource fork consisted of zero or more resources. Every resource had a resource type (4 bytes), a resource ID (4 bytes), and a value that could be any number of bytes. The operating system provided a Resource Manager to retrieve resources by ID. Within an application, every resource was given an ID and this was used to retrieve the bytes associated with the resource. The type provided information on how to edit the resource. Code fragments were also resources. An executable file consisted entirely of resources.
When the Mac moved away from its old operating system to the UNIX-based NeXT operating system, it inherited the .nib file structure. NIB [NeXT Interface Builder] files contain all of the resources necessary to build a user interface. The abandonment of the resource fork in favor of .nib files produced the same detachment problem. The Mac resolves this through bundles. A bundle is a folder or directory much like in any operating system. Inside a bundle is a hierarchy of files including user-interface resources. The Mac user interface to the file system treats bundles as special and makes them opaque to normal users. Files do exist in the bundle but they only get separated if someone writes special code to separate them.

The Microsoft .NET initiative created the notion of an assembly. An assembly gathers together all of the code and resources associated with an application. An assembly extends the programming language concepts of linker entry points to find other information besides code in the assembly.

In a number of modern IDTs, however, the mechanism for storing resources is code. Many IDTs are integrated with an IDE (Integrated Development Environment) such as Visual Studio or Eclipse. Because of this integration, the text editor for editing code and the IDT for editing interface designs are now combined into one tool. Figure 9.14 shows an interface being designed in Visual Studio. Figure 9.15 shows the C# declarations and constructor that are automatically generated when creating this form. A special method InitializeComponent() is added to the constructor automatically. Figure 9.16 shows some of the code automatically generated by the IDT in the InitializeComponent() method.

```
public class Form1 : System.Windows.Forms.Form
{
    private System.Windows.Forms.Button button1;
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;

    public Form1()
    {
        //
        // Required for Windows Form Designer support
        //
        InitializeComponent();
        //
        // TODO: Add any constructor code after
        // InitializeComponent call
        //
    }
    ...
}
```

Figure 9.15 – C# constructor code for a form

The button and radio button widgets are added as private variables of the Form1 class. This allows them to be readily found and accessed by both the application software and the IDT. Because of the way that the code is generated, it is just as easy for the IDT to find the widgets in this class file as it is in most resource files. The most complex parts of the generated code are found in InitializeComponent().
Every property specified in the IDT is encoded in InitializeComponent() as an assignment statement. As such, it is very easy to parse out the property values when the design is reloaded. Changing a property involves extracting the code from the file between region and endregion, generating a new version of InitializeComponent() and inserting the newly generated code back into place.

Visual Studio .NET is quite clean in the way its IDT connects with the code. Many such approaches are not as clean. The generated code is rather fragile and user modifications to the code can cause major damage from which the IDT might not be able to cleanly recover.

Globalization

One of the important purposes for resources is globalization of the user interface. Globalizing a user interface makes it possible to restructure the interface for a specific culture or language. By separating user-interface information from resources, you can localize the interface to a particular culture by modifying the resources rather than the code. The Eclipse IDE is very helpful in this regard by providing the “Externalize Strings” feature that locates all string literals and assists the programmer in changing those literals to resource references. The Visual Studio approach makes localization a little harder because the resources information is embedded in code. Localization teams can use Visual Studio to modify the resources visually, but these teams are now more intimately involved in the actual code. Globalization is discussed in more detail in Chapter 10.

BINDING EVENTS TO CODE

Manipulating properties is not enough when designing the user interface. Eventually, the interaction must connect with application code. The IDT must support this connection. One of the first attempts was the callback event model. Because every callback was associated with a string name, the IDT could manage events as string properties. The designer would simply enter the name of the desired callback into the associated property.

A second approach used in E2Win and JavaScript is to use an interpreted language for the user-interface implementation. If the underlying language is interpreted, then event handlers are text properties that have expressions in them. Whenever the event occurs, the associated property is evaluated as code.

A common approach in today’s IDTs is to generate code. Visual Studio .NET and C# provide good examples. Whenever the radio button in Figure 9.14 is selected, you want some code to be invoked so that other parts of the interface can be notified and updated. Using the reflection capabilities of C#, the IDE has searched for all event declarations in the implementation of RadioButton. They are listed in Figure 9.17. Remember that events and delegates are an integral part of the C# language. In addition to finding all of the events, reflection can also find all of the methods in the class whose argument type signature makes them candidates for each type of event. This allows the IDT to generate a list of such methods and place them in a pop-up menu.
Designers can then simply select acceptable event handlers from a list known to be correct. This was not possible in the old callback model because the set of available callbacks is not known until the application runs and registers them. If there is no appropriate method, the designer can request that a new one be generated, as shown in Figure 9.18. Notice that the IDE generates both the empty method with the correct declaration and the delegate assignment to the `CheckedChanged` event variable.

![Figure 9.17 - Visual Studio event list for RadioButton](image)

```csharp
private void InitializeComponent()
{
    this.radioButton1.Text = "radioButton1";
    this.radioButton1.CheckedChanged +=
        new System.EventHandler(this.radioButton1_CheckedChanged);
    
    
}

private void radioButton1_CheckedChanged(object sender, System.EventArgs e)
{
    // event code goes here
}
```

![Figure 9.18 - C# automatic generating of event handlers](image)

**SUMMARY**

Many aspects of a user interface design are visual. As such it is difficult to represent those design elements in code. Many times the interface designers are not programmers, but their special expertise is essential. Interface Design Environments provide tools that support visual specification and review of interactive designs in ways that are better suited to the problem and the skills of those involved. The layout model selected for a user interface design is strongly affected by the needs of a visual environment. Edge-anchored layouts are generally easier to manipulate visually than variable-intrinsic sized layouts.

When performing visual layout it is essential that the widgets appear as close as possible to their actual appearance in use. Using top-down event dispatching the redraw and resize events can be used for a widget while capturing the input events for use by the IDE.

Programming languages that support reflection are a great boost to the implementation of IDEs. Reflection simplifies the location of all widget classes in the code so that they can be used to populate a visible palette of tools. Reflection is also useful for identifying properties of widgets so that IDEs can visually edit the properties. The addition of annotation facilities to reflection models makes it easy to integrate visual information such as icons, colors and external names to be used in the IDE's user interface.

The most challenging part of IDE design is the connection between visual information and the code that must implement the controller and model. Reflection is very helpful here. Code generation has also been successful in tools like Eclipse or Visual Studio where the IDE is integrated into the tools used to edit and debug code.

**EXERCISES**

1. Why do edge-anchored layouts form a good compromise between variable intrinsic size layouts when you are trying to build a visual widget positioning tool?
2. How does the top-down event dispatching model described in Chapter 3 help to create a widget layout tool that uses live widget implementaions?
3. How is C# different from Java in the way widget properties are handled?
4. When an IDE is showing the properties of a widget, what can it do to handle the editing of property types that the IDE has never seen before?
5. What are some of the IDE architecture choices that keep resources such as colors and widget positions from getting separated from the code?
6. Why is it so hard for an IDE to make the connection between widgets and the code that actually handles the model?
Most of the people in the world are not North Americans or even European. Though English is the dominant language of trade, most people speak another language. Because human-computer interaction is designed to serve people, the creation of culture- and language-specific user interfaces is very important. This chapter does not cover all of the possible ways that a culture or language can impact software design, but it does cover the major ones. In some discussions of this topic, internationalization is abbreviated as "I18N."

When working on any software that deals with humans, you must consider the language and culture of the people involved. The gulf of evaluation occurs when the user translates information on the screen into an inappropriate mental model. That translation occurs in the context of the user’s language and culture. When creating an interactive consumer product, the user-interface designer must consider how the product will be used in Canada, France, China, and any number of other places. In some specialized cases, such as international flight and NATO military operations, the organization has purposely established language/culture standards that are uniform regardless of geographic or ethnic boundaries and, thus, internationalization might not be an issue. For virtually every other interface project, consideration of international issues is essential.

Internationalization itself is a misnomer when developing software. What you want is a globalized implementation for our user interface. A globalized interface is one that is prepared to be easily localized to a particular language and culture. A major issue in localization is adapting to the language. Not only do different languages have different words but they vary widely in how words are assembled into phrases and concepts. This can pose a challenge.