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 the user 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 may 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 we have built up a set of tools and techniques for programmers to create user interfaces. In this chapter we address 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 interactive tools that support the entire program development process. In this chapter we are 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 we need reflection or introspection[1]. Reflection is the ability at run-time to find out information about the program currently running. Until Java and C# were developed, most 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 will be briefly discussed.

The key reflection features we will need 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 tool-kit (see Chapter 5). The first published IDT was Buxton's MenuLay[2], shown in 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. Though the sketching tools were crude and the binding of events to callbacks very simplistic, this tool formed the inspiration for Apple HyperCard[3] and later Microsoft's Visual Basic[4]. In many ways layout design tools have not changed in twenty years. The paradigm is still 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.

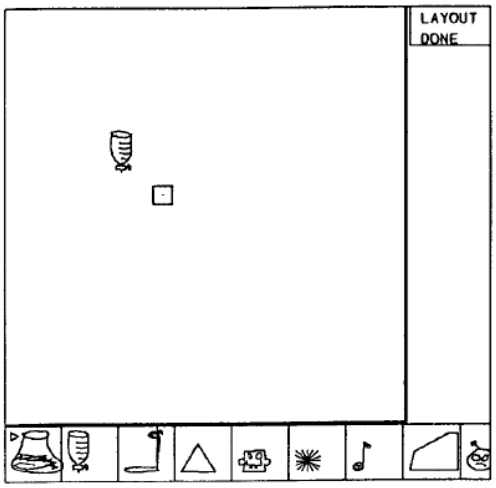

Figure 9.1 – MenuLay

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. We will briefly discuss 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 part 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[5] system introduced dynamic coordinates. These are essentially the edge-anchored layout mechanism found in C# 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.

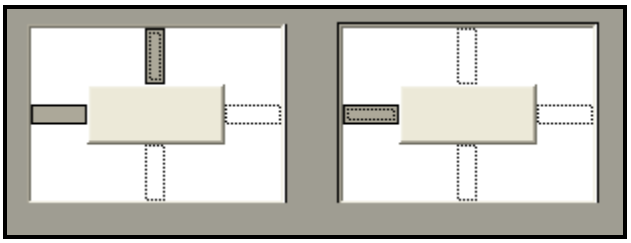

Figure 9.2 – Visual Studio .Net anchor property editor

MIKE took a different approach to the edge-anchored user interface. MIKE divided the screen into 9 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 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.

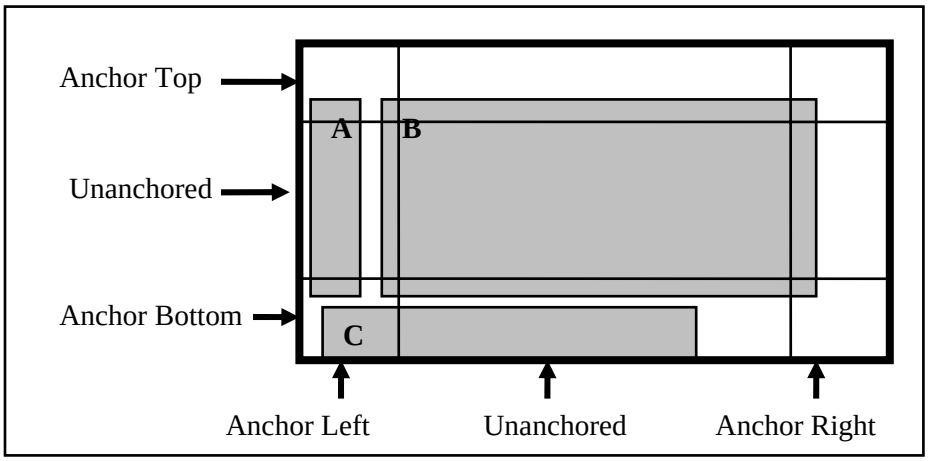

Figure 9.3 – Inferring anchors

Variable Intrinsic Size Layouts

Interactive layout design with intrinsic size is much more difficult. There are layout design tools such as Jigloo from CloudGarden[6] or NetBeans from Sun[7]. These tools do not have the same ease of use because their model is not geometric. The model for variable intrinsic size is a tree of widgets with layout managers and minimum, preferred and maximum sizes. These layouts are more naturally designed programmatically.

The tools that do exist for interactive design of intrinsic size layouts face two difficulties. The first is that designers expect widgets to stay approximately where they are put. However, it is the layout manager and preferred size, not the original widget placement that controls layout. This conflict of control can be very frustrating to designers. It is also difficult for those who create the tools because designers will specify placements and the tool must do the "nearest appropriate thing". This can lead to combinations of settings that can create bizarre behaviors.

A second problem 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 tool bars", "Menu bar" and "File" menu header all have the same geometric position for their top edge. The left edge of "Menus and tool bars" is completely occupied by the left edges of "Menu bar", "Tool bar 1" and "Tool bar 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 may 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.

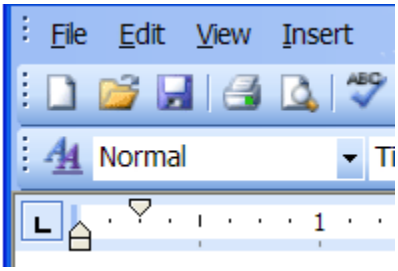

Figure 9.4 – Toolbars and menus

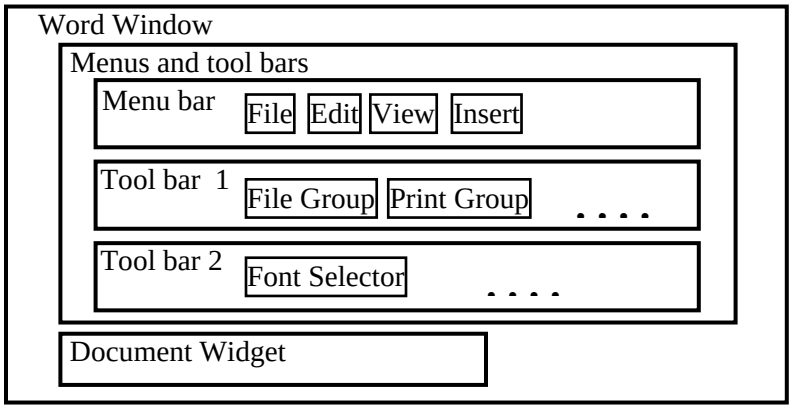

Figure 9.5 – Menu and toolbars widget tree

Drawing with live widgets

The layout of drawn icons and text labels is still an important part of user interface design, but in modern systems it is the layout of widgets that is most important. Figure 9.6 shows two widgets being positioned using Visual Studio. These two widgets have had various properties set to change their color and font. The layout design tool must support any widget that has been added to the configuration including 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 we want. What we 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 will pass them on to the child widgets so that they will appear as they should. With mouse events, however, the design tool will retain 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.

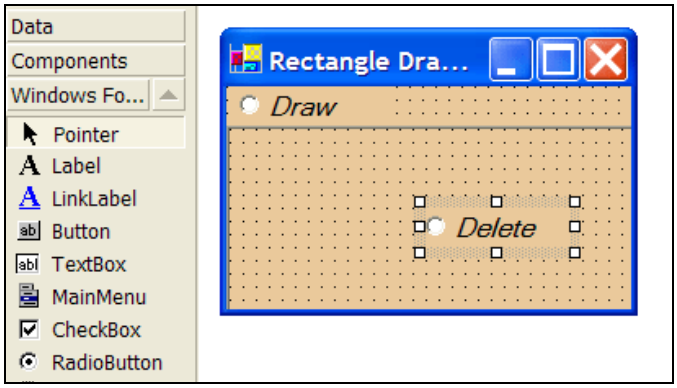

Figure 9.6 – Laying out live widgets

Finding all of the widgets

We want our 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 we 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 example Java code for retrieving a widget given a class name.

```
Widget getNewWidget(String widgetClassName)
{
    try
    {   Class widgetClass= Class.forName(widgetClassName);
        Object obj = widgetClass.newInstance();
        return (Widget)obj;
    } catch (Exception E)
    {   System.out.println(widgetClassName+" is not a Widget class");
        return null;
    }
}
```

Figure 9.7 – Creating a widget from a class name in Java

The C# equivalent shown in figure 9.8 is slightly more complicated. The Type class instead of the Class class is used. From the Type class one retrieves a ConstructorInfo object from which an instance of the class can be created.

```
Widget getNewWidget(String widgetClassName)
{
    try
    {   Type widgetType=Type.GetType(widgetClassName);
        ConstructorInfo constInfo=widgetType.GetConstructor(Type.EmptyTypes);
        Object constArgs[]=new constArgs[0];
        Object obj = constInfo.Invoke(constArgs);
        return (Widget)obj;
    } catch (Exception E)
    {   return null; }
}
```

Figure 9.8 – C# construction of new widgets

There still remains the problem of the menu of widgets on the left side of figure 9.6. There are several approaches to this problem. A simple approach is to give the Widget class two methods getMenuName() and getMenuIcon(). A new widget implementation would provide the needed information through these methods. A second approach is to augment our configuration file so that each widget class name is accompanied by the menu name and the name of a file containing the desired icon.

A third approach for providing this information is to use a reflection technique called annotation. In C# these are referred to as attributes. Annotations or attributes are pieces of data that can be attached to declarations in the source code and bound at compile time to the reflection information. Using attributes we can attach the menu name and menu icon information to the Widget class. Figure 9.10 shows how this is done in C#. A special class WidgetInfo is defined as a subclass of Attribute. It is given members and a constructor to store the information. When the class MySpecialWidget is defined, it is preceded by the

attribute declaration that provides the menu name and menu icon information. In the widgetMenuName() method this information is retrieved from the Type information of a widget so that it can be used to display the menu.

```
public class WidgetInfo : Attribute
{
    public String menuName;
    public String menuIconFileName;
    public WidgetInfo(string mName, String mIconFile)
    {   menuName=mName;
        menuIconFileName=mIconFile;
    }
}

[ WidgetInfo("My Special Widget", "icons\special.gif") ]
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.GetType().GetFullName();
}
```

Figure 9.10 – 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 tool-tip 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 we want to cast 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 important. This is further complicated by the fact that new properties may have new data

types. Specifying such property values also complicates the IDT architecture. When a design is complete and all of the properties 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 will be discussed more extensively in chapter 10, but its implementation foundation will be discussed here.

Figure 9.11 shows the Jigloo[8] 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. There are a variety of kinds of properties including booleans, fonts, colors, lists of choices, icon file names and integer numbers. An IDT must be able to adapt to whatever properties a widget may have and allow the user to edit those properties.

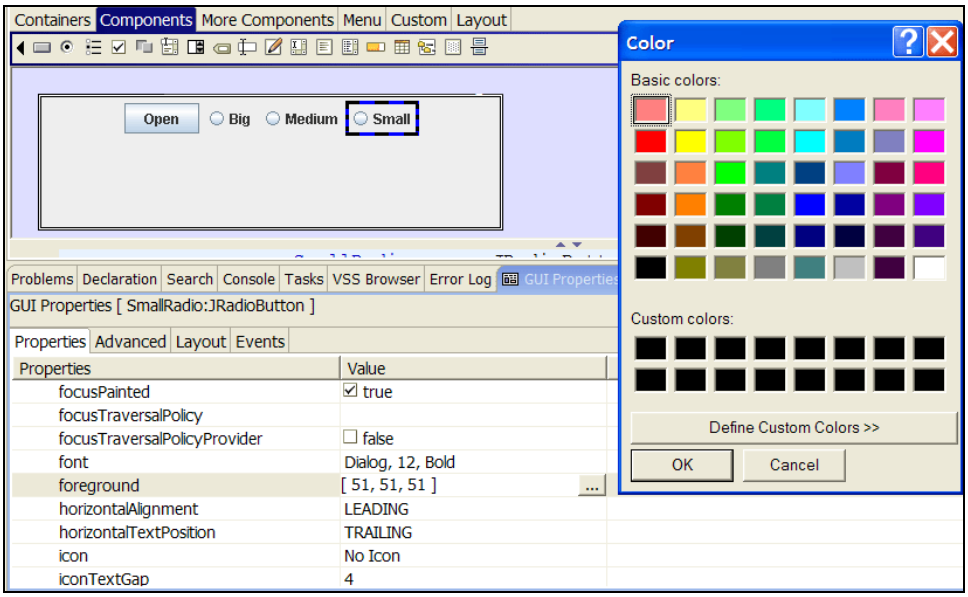

Figure 9.11 – Jigloo properties for a radio button

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:

type get*PropertyName*();

void set*PropertyName*(*type* newValue);

Either of these two mechanisms can be handled using reflection. The Java Class class has a method to 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 one 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 one can construct the property list shown in figure 9.11. 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.

There are four problems with this approach to editing properties. The first is parsing property values typed in by a user. The second is when the property names are defined for the consumption of programmers rather than designers. The third problem is when a property is intended for software use, but not to be exposed to the designer interface. Lastly there is the problem 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 will use the 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 we can require a WidgetProperty annotation that includes the public name of the property. This resolves both the naming and the interactive access problems. A StringParse annotation can name a method that will translate a string into an appropriate property value. A PropertyEditor annotation can name the class of a widget that can edit a property's value and produce a new value.

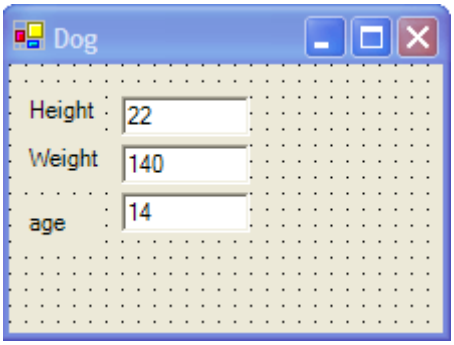

Figure 9.12 – User interface being designed

Storing Resources

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 9.12 shows a user interface design in progress. The model for the application being designed is shown in figure 9.13. The model that the IDT is working on is shown in figure 9.14. The application will be editing information about dogs while the IDT is editing information about widgets.

```
public class Dog
{
    int heightInches;
    int weightPounds;
    int ageMonths;
}
```

Figure 9.13 – Dog model

```
public class Form extends Widget
{
    WidgetDescriptor [] widgets;
    String title;
    Color background;
    Color foreground;
    . . . .
}
public WidgetDescriptor
{
    String widgetClass;
    Rectangle bounds;
    int edgeAnchors;
    WidgetProperty [] properties;
    . . . .
}
public WidgetProperty
{
    String propertyName;
    Object propertyValue;
}
```

Figure 9.14 – IDT model

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. There are several ways in which this can be done. 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[9] designed for X-windows[10] 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. Without the resources created by the IDT the application has no information about how to structure the interface. The Apple Macintosh resolved this 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 we 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. There are files 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 (Interactive 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 9.15 shows an interface being designed in Visual Studio. Figure 9.16 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 9.17 shows some of the code automatically generated by the IDT in the `InitializeComponent()` method.

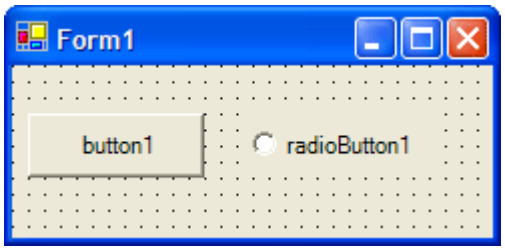

Figure 9.15 – User interface design

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().

```
public class Form1 : System.Windows.Forms.Form
{       private System.Windows.Forms.Button button1;
        private System.Windows.Forms.RadioButton radioButton1;
        /// <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 9.16 – C# constructor code for a form

```
#region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
        this.button1 = new System.Windows.Forms.Button();
        this.radioButton1 = new System.Windows.Forms.RadioButton();
        this.SuspendLayout();
        //
        // button1
        //
        this.button1.Location = new System.Drawing.Point(8, 24);
        this.button1.Name = "button1";
        this.button1.Size = new System.Drawing.Size(88, 32);
        this.button1.TabIndex = 0;
        this.button1.Text = "button1";
        //
        // radioButton1
        //
        this.radioButton1.Location = new System.Drawing.Point(120, 24);
        this.radioButton1.Name = "radioButton1";
        this.radioButton1.Size = new System.Drawing.Size(96, 32);
        this.radioButton1.TabIndex = 1;
        this.radioButton1.Text = "radioButton1";
        //
        // Form1
        //
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
        this.ClientSize = new System.Drawing.Size(240, 86);
        this.Controls.AddRange(new System.Windows.Forms.Control[] {
            this.radioButton1,
            this.button1});
        this.Name = "Form1";
        this.Text = "Form1";
        this.ResumeLayout(false);

    }
#endregion
```

Figure 9.17 – 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 may 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 into resources one 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. Globalization will be 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 EZWin[11] 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# provides a good example. Whenever the radio button in figure 9.15 is selected we 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 IDT has searched for all event declarations in the implementation of `RadioButton`. They are listed in figure 9.18. Remember that events and delegates are an integral part of the C# language. In addition to finding all of the events, the 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 popup menu next to the event name. Designers can then simply select acceptable event handlers from a list known to be correct. This was not possible in the old callback model. If there is no appropriate method the

designer can request that a new one be generated as shown in figure 9.19. Notice that the IDT generates both the empty method with the correct declaration and the delegate assignment to the CheckedChanged event variable.

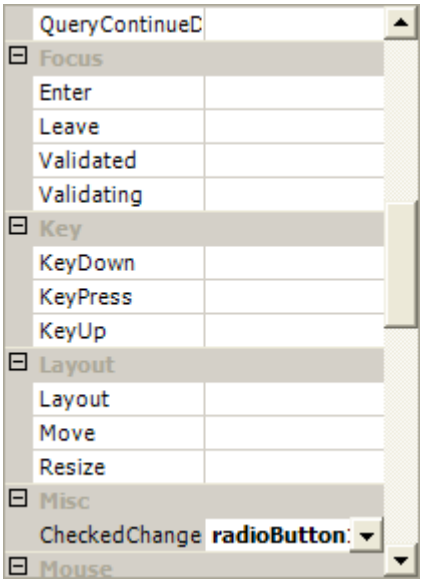

Figure 9.18 – Visual Studio event list for RadioButton

```
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 9.19 – C# automatic generating of event handlers

Exercises

1. Why do edge anchored layouts form a good compromise between variable intrinsic size layouts when one is trying to build a visual widget positioning tool?

2. How does the top-down event dispatching model described in chapter 3 help us to create a widget layout tool that uses live widget implementations?
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?

---

[1] Kiczales, G., des Rivières, J., and Bobrow, D. G. *The Art of Metaobject Protocol*. MIT Press (1991).

[2] Buxton, W., Lamb, M. R., Sherman, D., and Smith, K. C. "Towards a Comprehensive User Interface Management System." *Computer Graphics and Interactive Techniques (SIGGRAPH '83)*, ACM, (1983)35-42.

[3] Goodman, D., *The Complete Hypercard 2.2 Handbook*, iUniverse, (1998).

[4] *Microsoft Visual Basic .Net Standard 2003*. Microsoft Software, (2003).

[5] Olsen, D. R. "MIKE: the Menu Interaction Kontrol Environment." *ACM Trans. Graph.* 5, 4 (Oct. 1986), 318-344.

[6] http://cloudgarden.com/jigloo/

[7] Boudreau, T., Glick, J., Greene, S., Woehr, J., and Spurlin, V., *NetBeans: The Definitive Guide*, O'Reilly Media, Inc. (2002).

[8] http://cloudgarden.com/jigloo/

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

[10] Scheifler, R. W. and Gettys, J. The X Window System. *ACM Trans. Graph.* 5, 2, 79-109. (Apr. 1986)

[11] Lieberman, H. "There's More to Menu Systems than Meets the Screen." *Computer Graphics and interactive Techniques (SIGGRAPH '85)*, (1985) 181-189.