5. The Implementation of Components

5.1. Prerequisites and Objectives

Before starting this chapter you should have:

1. A knowledge of component categories and how they are used to break an application into components.
2. Some familiarity with object diagrams, UML or some other notation.

After completing this chapter you will have:

1. A knowledge of how to design a component interface.
2. A knowledge of the passive design philosophy and how it relates to component interface design.
3. The ability to plan and document a component interface.
4. Optionally, a knowledge of how to use the Visual Studio ActiveX design tools.

5.2. Introduction

Object oriented design is an integral part of component development, because components share many features with objects, and can be used in much the same way. The main difference is that components are generally broader in scope, and have more rigidly defined interfaces. The breakdown into components is typically done only at the highest level. Because components are separately compiled programs, the breakdown is generally not hierarchical. (This may change in the future.)
Component level design and object oriented design are actually independent concepts, but it has been found to be useful to tie the concept of component level design to that of object oriented design. The most important marriage between object oriented design and component level programming is the encapsulation of component functionality in a single support class. Each instance of a component has its own support object, which encapsulates all instance-specific data, and provides the support functions to handle both user and program input. The encapsulation of data and functionality is necessary to support multiple instances of a component. This encapsulation was necessary for the earlier non-object-oriented technologies as well, but these technologies solved the problem in less elegant ways.

In more recent technologies, the support class is an integral part of the component. This class inherits much of its functionality from standard operating system classes, and contains the functions and data items necessary to implement the component. In reality one does not design a component, one designs the support class of the component. The standard functionality inherited by the support class will perform the low-level operations necessary to integrate the component with its environment. When a component is instantiated, standard functionality built into the component and its environment will create an instance of the support class to encapsulate the details of the component instance. The instances of the support class are called support objects. Figure 5.1 shows the relationship between components and the support class. Although there is additional required functionality beyond that of the support class, this functionality is generally invisible to the component developer.
As we will demonstrate in later chapters, the design of most types of components involves a significant amount of object-oriented design beyond the design of the support class. In most cases, these classes become aggregates or friends of the support class. Some components serve as mere wrappers for objects, and others require intensive object-oriented design to implement their functionality in a manageable way. No specific object oriented design methodology is recommended, but it is important to have some familiarity with formal design techniques before proceeding.

5.3. **Properties, Methods, and Events**

The interface of a component consists of a set of properties, a set of methods, and a set of events. To document the interface we need three different types of specifications,
property specifications, method specifications, and event specifications. Before
proceeding with the details of these specifications, it is important to establish some
general rules about how properties, methods, and events are used.

Although a property has much in common with a variable, it is important to
remember that it is not a variable. The component is capable of monitoring both read and
write accesses to the property, and is capable of responding to a read or write request in a
complex way. For example, assigning a value to a FileName property may cause a file to
be written to disk. When documenting a property it is important do document the actions
that will be taken in response to read and write requests. Properties can be read-only,
write-only, or read-write. Furthermore, it is possible to restrict access to a property to
make it design-time only, or run-time only. A design-time only property can have a value
assigned to it by a visual design tool such as Visual Basic, but a value cannot be assigned
to the property at run-time. A run-time only property will not be visible to the
programmer at design time, but it will be possible to read it or change its value at run-
time. The formal specification of the property must contain all restrictions on the usage of
a property.

The value of a property must be a simple type such as Short Integer or Double
Floating Point, or an array of simple types. (This is a concession to some of the most
popular technologies, which do not permit objects as property types. This is a severe
restriction that does not exist in all technologies, will probably not exist in any future
technology.) If it is necessary to send a set of coordinated values to a component, a
method, with multiple arguments, must be used. However, methods cannot be used at
design time. If it is necessary to supply a coordinated set of values at design time, the
component must be designed to tolerate sets of inconsistent values, because these values must be supplied one at a time. For example the *Pentomino* component of Chapter 6 has two properties that permit the user to specify the dimensions of the playing surface. The product of the \(x\) and \(y\) dimensions must be 60, but the component cannot raise an exception when a set of inconsistent values has been specified, because the values must be specified one at a time.

Because properties can trigger complex behavior, it is necessary to make a careful decision about whether a particular feature is to be implemented as a property or as a method. There is no hard and fast rule for making such a decision, but there are some guiding principles that will help.

One should consider replacing write-only properties with single-argument methods. A write-only property will not be visible at design time, because the visual display of a property requires that its value be readable at design-time. If the value assigned to the property is retained by the component after processing and used by the component in other ways, then a property may be more appropriate implementation than a method. However, if the value is used immediately and then discarded, or not used at all, a method is a more appropriate implementation. By the same token, properties that are both read-only and run-time-only can be replaced by a method with zero arguments.

If you find yourself creating “GetValue” and “SetValue” methods for simple values, you should consider using a property instead. (Except, of course, for those technologies where properties are specified using Get and Set functions.) It is not necessary for the value of a property to be represented explicitly in the support class. Property values can
be computed when they are requested. The choice between property and a method will usually be obvious.

Although events and methods are quite different from one another, people who are just learning about component technology often get confused as to which is which. A method is a function implemented by the component that is called by the container at runtime. An event is a function implemented by the container that is called by the component at runtime. The container always has the option of doing nothing in response to an event. An event has no behavior defined by the component. If you find yourself thinking about the behavior of an event, you are not thinking about an event. You are thinking about a method. As far as the component is concerned, an event is nothing more than a function call. The body of the function is in the container, not in the component. The primary use for events is to inform the containing program that something important has occurred. The classic example of an event is the “Click” event provided by the standard button component. The button calls the click-event function to inform the host program that the button has occurred. The Visual Basic programmer (for example) can provide a function body for the click event to perform useful work when the button is clicked. Figure 5.2 summarizes the characteristics of properties methods and events. This information can be used to help choose an appropriate implementation for each interface item.
<table>
<thead>
<tr>
<th>Property</th>
<th>Method</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can hold simple data items</td>
<td>Valueless</td>
<td>Informs User</td>
</tr>
<tr>
<td>Passes simple data items to component</td>
<td>Usually triggers complex behavior</td>
<td>Passes parameters to User</td>
</tr>
<tr>
<td>Retrieves simple data items from component</td>
<td>Passes several coordinated data items to component</td>
<td></td>
</tr>
<tr>
<td>Available at design time</td>
<td>Retrieves simple data items from component</td>
<td></td>
</tr>
<tr>
<td>Design time assignments are persistent</td>
<td>Not available at design time</td>
<td></td>
</tr>
<tr>
<td>Sometimes triggers complex behavior</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.2. Property, Method, and Event Characteristics.

5.4. The Passive Implementation Style

In this book we recommend a passive style of implementation. This implies that we want to minimize the direct interaction between the program user and the component. Except for Editor components, we believe that components should not directly process mouse or keyboard input. If the user clicks on the component, the mouse-click, and possibly its position, should be reported by an event, but beyond that the component should take no action. If the user presses the Delete key, a DeleteRequested event should be issued, but no direct action should be taken. Over several years of component based development, we have observed that components that use the passive implementation style are both easier to create and easier to use than their active counterparts.

The passive implementation style is actually a set of rules that should be followed when developing a component. In the remainder of this section, we will discuss the individual rules, and their impact on component design.

The first rule is to pass mouse-clicks through the component unprocessed. One can process the input coordinates to determine the part of the display that has been clicked,
but other than this, the component should take no action. The component container may have several modes of operation that are unknown to the component. The mouse-click may mean something different depending on the mode. Passing the click through unchanged permits these modes to work properly. A component that actively processes mouse clicks can be used only with the modes that have been designed into it. In most cases, the glue logic will process the mouse-click by performing some action on the component.

To see how this rule works in practice, imagine a tic-tac-toe component. When the user clicks on a square, an X should be drawn in the square. However the component does not do this correctly. Instead it issues an event indicating which square has been clicked. The glue logic intercepts the event and calls a component method to draw the X.

This rule does not apply to Editors, which must interact heavily with the user.

The second rule is to avoid direct implementation of menu shortcuts. This especially applies to the Cut, Paste, Copy, and Delete operations. These are menu shortcuts and should be directed through the menu system by the operating system. Your component is not part of the menu system. This rule applies to all components, even editors. If you find it absolutely necessary to process some menu shortcuts, provide a property that can be used to turn off this processing. If you need Cut, Paste, Copy, and Delete operations, provide a method for each. These methods must be the only mechanisms for performing these operations. Cut, Paste, Copy, and Delete operations trigger complex actions which should be implemented at a single site within the component.

The third rule is do not issue error messages. All errors must be reported by using events or return codes. In some cases it is possible to use a status-code property in place
of a return code. If you violate this rule, your component will be much more difficult to internationalize. Any English text will have to be translated and replaced with text in some other language. It is possible to do this without recompiling the component, but the procedures that one must go through to enable this are beyond the scope of this book. Furthermore, what you consider to be an error might not be considered to be an error by the user. Suppose you wanted to write a program that would sort files into three categories, C-Code, PASCAL-Code, and Other. You could use a C-Compiler component and a PASCAL-Compiler component to test for C-Code files and PASCAL-Code files. Since it is not an error for a file to be in the category Other, error messages from the compiler components would be an unwanted annoyance. A parser error merely indicates that a file is not of a certain type. It does not indicate an error.

The fourth rule is do not query the user for input. If it is necessary for you to open a file, implement a FileName property, and open a file whenever a new value is assigned to this property. Do not pop up an open-file dialog box. All component input should come through the Property/Method interface (except for Editor components).

The fifth rule is don’t ask the user any questions. This is a special case of the third and fourth rules. Suppose you have implemented a word-processing control, and you want to provide it with a “Change All With Confirmation” command. When you encounter a match for the input pattern, do not pop up a dialog box saying “Do you want to change this instance?” Instead, issue an event and let the glue logic pop up the dialog box.

The sixth and final rule is limit keyboard input to shortcuts. In other words, if it is possible for the user to perform a task using the keyboard, it should be possible to
perform the same task using properties and methods. Specifically, if the left arrow can be
used to scroll to the left, then there should be a **ScrollLeft** method that does the same
thing. The keyboard input must be a shortcut for something that can be done through the
property/method/event interface.

Although these six rules govern most situations that you will encounter, it is
important to remember that the passive implementation style is a design philosophy, not a
set of rules. It is important to follow this design philosophy, even if a particular situation
is not covered by the rules.

### 5.5. Formal Specifications

In any mid to large-scale software design project it is important that certain formal
specifications be completed before one proceeds with the coding. Component level
design is no exception to this rule. We will not invent a new type of formal specification
for component internals, but will rely instead on those provided by existing object-
oriented design methodologies. However, we do recommend a set of formal
specifications be completed to describe the interface of the component. Insofar as it is
possible, these specifications should be completed before proceeding with the
implementation.

#### 5.5.1. Properties

Figure 5.3 shows the property design table, which we will use to give the formal
description of properties. Column 1 gives the name of the property, column 2 gives the
type, the default value, and any restrictions on the property value. The type is given in
narrative form rather than using a language-specific type specifier. The third column
contains a narrative description of the property’s function, along with any restrictions on its use. If there are no restrictions, the statement “Restrictions: None” may be omitted. A property table entry should be completed for every property before it is implemented.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Long Integer</td>
<td>Contains the value currently displayed by the component</td>
</tr>
<tr>
<td></td>
<td>Default = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Restrictions: None</td>
<td></td>
</tr>
<tr>
<td>ArraySize</td>
<td>Long Integer</td>
<td>When assigned a value, the internal array is destroyed and replaced with</td>
</tr>
<tr>
<td></td>
<td>Default = 20</td>
<td>a new array. The value of this property determines the size of the array.</td>
</tr>
<tr>
<td></td>
<td>Minimum = 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum = 50</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Restrictions: Run Time Only</td>
<td></td>
</tr>
<tr>
<td>QueueSize</td>
<td>Long Integer</td>
<td>This property is used to report the size of the internal queue. The queue</td>
</tr>
<tr>
<td></td>
<td>Default = 0</td>
<td>size changes when the Push and Pop methods are called. These two methods</td>
</tr>
<tr>
<td></td>
<td></td>
<td>are responsible for maintaining this value.</td>
</tr>
<tr>
<td></td>
<td>Restrictions: Read Only</td>
<td></td>
</tr>
<tr>
<td>Caption</td>
<td>String</td>
<td>The caption will be displayed at the top of the control window both at</td>
</tr>
<tr>
<td></td>
<td>Default = empty string</td>
<td>run time and design time.</td>
</tr>
<tr>
<td>List</td>
<td>Long Integer</td>
<td>Allows the programmer to store up to five values and retrieve them later.</td>
</tr>
<tr>
<td></td>
<td>Array of size 5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Default = none</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.3. The Property Design Table.

5.5.2. Methods

Figure 5.4 shows the Method description diagram that will be used to document all methods. The upper portion of the diagram is used to document the function itself by giving its name, return value, and a narrative description of its function. The second part of the table describes each argument. Column 1 gives the ordinal position of the
argument in the argument list. If these numbers are omitted, the arguments are assumed
to be described in the order in which they must be specified. Column 2 gives the name of
the argument, column 3 gives the type in narrative form, and column 4 gives a
description of the argument. Any restrictions on the value of the argument must be listed
in the Type column. A method description must be completed for each method before it
is implemented.

<table>
<thead>
<tr>
<th>Name</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Value</td>
<td>Double Float</td>
</tr>
<tr>
<td>Description</td>
<td>Computes the Pythagorean distance between two points.</td>
</tr>
<tr>
<td>Arguments</td>
<td>Name</td>
</tr>
<tr>
<td>1</td>
<td>FromX</td>
</tr>
<tr>
<td>2</td>
<td>FromY</td>
</tr>
<tr>
<td>3</td>
<td>ToX</td>
</tr>
<tr>
<td>4</td>
<td>ToY</td>
</tr>
</tbody>
</table>

Figure 5.4. A Method Description.

5.5.3. Events

Figure 5.5 shows an event description diagram. Except for two lines in the upper
section, the diagram is the same as the Method Description Diagram. The Event
Description Diagram has no return value type, but has a Triggers description instead.
(Although some technologies permit return values for events, this is not universal.) In the
Triggers description it is necessary to document the precise conditions under which the
event is fired. The description of the event should describe the event and its intended
usage.
### Event Description

<table>
<thead>
<tr>
<th>Name</th>
<th>SelectSquare</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Provides the row and column of a selected square.</td>
</tr>
<tr>
<td>Triggers</td>
<td>Triggered when a user clicks on a square in the component window, or when the TouchSquare method is executed</td>
</tr>
<tr>
<td>Arguments</td>
<td>Name</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>Row</td>
</tr>
<tr>
<td>2</td>
<td>Column</td>
</tr>
</tbody>
</table>

**Figure 5.5. An Event Description.**

### 5.6. Individual Component Development

In the following chapters we will give design methodologies and examples for each different category of component. Many of these chapters are independent of one another, but some of them are closely interrelated, and should be read in a specific order. In particular, Chapter 7 introduces the concept of graphical editors through the use of an example called the simple graphical editor. The underlying object structure created to support the simple graphical editor is complex and can easily serve as a case study in object oriented design. This object is intended to be passed from component to component, and thus is used by a number of different components introduced in a number of different chapters. The main object is called the CGraphicList object. This object contains a number of aggregated object, and provides functions to support adding, deleting, and modifying graphical elements. It also supports an undo facility and a serialization facility. These facilities are used in a number of different components.

Figure 5.6 illustrates the usage of this object by the various different components, and also indicates in which chapter each component is presented. Like the example of the Prospect Management Program given in Chapter 4, the SGE (Simple Graphical Editor)
components show how to develop an application using a series of interrelated components.

Figure 5.6. Simple Graphical Editor Components.

Chapters 7, 8, and 9 are exclusively devoted to SGE examples, while Chapters 10, 11, and 12, have a substantial number of other examples. (Chapter 7 contains the introductory material for the SGE family of components, and it is important to read this chapter first.) Because the overall structure of the CGraphicList object is developed over several chapters, we include a preview of the entire structure in Figure 5.7.
In Figure 5.7, the diamond shape signifies aggregation or inclusion, while the arrowhead signifies inheritance. The left hand side of Figure 5.7 is concerned with the representation of graphical objects, while the right hand side is concerned with the undo facility. Both of these concepts are explained in Chapter 7. The objects SerRectangle and SerCircle are serial forms of the CRectangle and CCircle objects. The SerRectangle and SerCircle objects are suitable for writing to persistent storage, while the CRectangle and CCircle objects are not. Serialization and persistent storage are discussed in Chapter 9.
Although a family of components is an extremely powerful development tool, independent components designed to perform a single task without cooperation from other components are equally powerful. Much of the material of Chapters 10, 11, and 12 is devoted to independent components, while Chapter 6 and Chapters 13-18 are entirely devoted to independent components.

5.7. ActiveX Development

Although there are many different component technologies, we will focus on a single technology, ActiveX, in the remainder of this book. To describe the process of component development, it is necessary to give numerous examples. To show the code of an example, we must first select a technology in which to implement it. (The code illustrated in this book is all cut and pasted from running examples. We do not believe in creating “abstract code” merely for the purpose of illustrating a point.) Since all current technologies share the same sorts of features, concentrating on a single technology is not restrictive. To convert our examples into JavaBeans, for example, it is simply a matter of translating our examples into their JavaBeans equivalents. If one must select a single technology, ActiveX is a reasonable choice. It is the leader in the commercial world, and it allows us to create components that can be used in a wide variety of different applications. An ActiveX control can be used in the Visual Basic, Visual C++, Delphi and C# languages. It can be placed in a web page and distributed automatically through the Internet. It can be placed in a Microsoft Word document, a PowerPoint presentation
or an Excel spread sheet. In addition, ActiveX controls can be developed in many different programming languages. Furthermore, because of the large base of existing ActiveX components, it is unlikely that they will be replaced by newer technologies at any time in the near future.

There is also a price to be paid for this choice. An ActiveX control is complicated, so much so that we must automate much of the development process through the use of development tools. This ties the discussion to a specific set of tools, and may tend to give the impression that these tools are the only way to create components. The reader is cautioned that there are many other tools that provide the same functionality as those described here.

The specific set of tools that we will use are the MFC and ATL development packages that have been integrated into the Visual C++ development platform. (As of this writing, version 6.0 is the current version of this platform.) We will use only the MFC tools in this book. ATL is beyond the scope of this book, but we provide some ATL projects on the CD ROM.

5.7.1. MFC

The term MFC stands for Microsoft Foundation Classes. The MFC package provides wrappers for most of the basic Windows operating system facilities. In particular, it provides a wrapper for ActiveX controls. To create an ActiveX control, start Visual C++, and select the New command from the File menu. This will display the dialog box shown in Figure 5.8. Type a name in the Project Name field and click OK.

Next, you will be shown the dialog box pictured in Figure 5.9. Don’t do anything with this, just click Next. This will give you the dialog box shown in Figure 5.10. If you
want a visible control, do nothing, just click *Finish*. If you want an invisible control, click Invisible at Run-Time so that it is checked, and uncheck Activates When Visible. Then click on *Finish*. You may now compile your ActiveX control, although to make it interesting, you will need to add a little bit more to it.

Figure 5.8. Starting a New MFC Project.
Figure 5.9. MFC ActiveX Control Wizard, Page 1.
You add properties methods and events to your component using the Class Wizard which is accessed from the View menu. (See Figure 3.7 for a picture of the Class Wizard.) The Automation tab of the Class Wizard is used to add properties and methods to the control, while the ActiveX Events tab is used to add events. Click on the proper button and follow the instructions to create Properties, Methods, and Events. The Message Maps tab is used to intercept user input. In most cases the Class Wizard can be used without explicit instructions.

The function of the Class Wizard, among other things, is to add functions to the support class of your ActiveX control. You implement your ActiveX control by supplying the bodies of these functions.
One function that is automatically supplied for you is the OnDraw function. This function is called any time the window needs to be redrawn, and whenever your component calls the CWnd::Invalidate function. The OnDraw function has three parameters, the drawing context of the control window, a rectangle describing the entire window, and a rectangle describing the part of the window that must be redrawn. MFC Drawing functions must be used with the drawing context to draw the window.

The process for creating an ActiveX control in VS.NET is quite similar to the procedure used in Visual C++ 6.0. One goes to the File Menu, selects the New command and the Project sub-command. This will display the dialog box given in Figure 5.11. From this dialog box, select Visual C++ Projects from the left window, and MFC ActiveX Control from the right window. Type the project name in the Name box, and click OK.

![New Project dialog box](image)

**Figure 5.11. Creating an ActiveX in VS.NET.**
The VS.NET ActiveX wizard is essentially the same as that for Visual C++ 6.0, but it is organized somewhat differently. Figure 5.12 shows the first page of the wizard. The subsequent pages can be accessed by clicking on the phrases “Application Settings”, “Control Names,” and “Control Settings”. The “Application Settings” page corresponds to page 1 of the VC++ 6.0 wizard, while the “Control Settings” page corresponds to the second page of the VC++ 6.0 wizard.

As noted in Chapter 3, in VS.NET, a number of the class wizard functions have moved into the class view window. Others have moved into the property window. See Chapter 3 for information on creating properties, methods, and events. To add functions
of other types, select the class C<YourName>Ctrl, and use the properties window to add the desired function. Figure 5.13 shows how to add a mouse-click event handler to a component named SampleStuff, while Figure 5.14 shows how to add an override to the same component. Note the row of pictures below the name of the class. To add message handlers, click on the 5th picture from the left, to add an override, click on the 6th picture from the left. In both cases, you add the desired function by clicking on the suggested function name.

![Figure 5.13. Adding A Message Handler in VS.NET.](image)

Figure 5.13. Adding A Message Handler in VS.NET.
5.8. Conclusion

For components, the most important implementation skill is determining when to use a property, and when to use a method for a particular interface feature. The guidelines presented above will help, but some practice is necessary. The passive implementation style is an important tool for simplifying component development, because much work is passed off to the glue logic rather than being performed by the component itself. Surprisingly, this implementation style also tends to simplify the glue logic, because the programmer does not have to conform to a set of conventions established by the component.

Documentation and planning is important at all phases of application design. The most important formal specifications for a component are the interface specifications that document the properties, methods, and events. These interface specifications should be
completed at the same time as other design documents such as object diagrams and state diagrams.

Once you have mastered the skills of this chapter, and of Chapter 4, you are ready to begin developing your own components.

5.9. Exercises

1. You are designing a component that needs to read a configuration file. Two different instances of your component may need to use two different files. The configuration file is processed during initialization, but it is also possible to switch from one configuration file to another while a component instance is running. Discuss the best way to implement these features.

2. You are designing a component that needs to read several different files while it is running. These files will be read one at a time, and each file will be fully processed and closed before reading the next. If a specified file cannot be opened, the user should be notified. Discuss the best way to implement these features.

3. You are designing a component that implements a dynamic one-dimensional array. Initially each instance of the array has a default size. (There is one array per component instance.) The size of the array can be changed dynamically at run time. Using formal design documents, show how the size of the array will be specified, and show how to access and modify the individual elements of the array.

4. You are designing a random greeting component. The user must initialize a component instance by assigning a number of strings to it. These strings will be stored in an internal list. When the user requests a greeting, a random choice will be
made from the internal list of strings and that string will be returned to the user. Using the formal design documents, show how these features should be implemented.

5. You are designing a Parcheesi game simulator that shows the board, the pieces on the board, and the dice. When a player makes a move, he/she will click on a button marked “Roll” to roll the dice. (Note that in Parcheesi, two dice are rolled, and the player may move one piece the total number of spaces indicated on both dice, or two pieces, one the distance shown on the first die, and the second the distance shown on the second die.) To make a move the player will first click on a piece to select it, and then click on a die to move the piece that distance. The moved piece will remain selected so that it can be moved again. Two custom components are required, one to represent the dice and one to represent the board. Using the formal design documents, describe the portions of the interfaces that permit rolling the dice and making moves.