

4. Component-Based Application Design

4.1. Prerequisites and Objectives

Before starting this chapter you should have:

1. A rudimentary knowledge of Component Level Programming.
2. Some knowledge of object oriented programming.

After completing this chapter you will have:

1. A knowledge of how to divide an application into components.
2. A knowledge of component categories.
3. The ability to use component categories as a design tool.

4.2. Introduction

A modern GUI-based program consists of many different parts. At the user-interface level there is at least a main window and a menu bar. There may also be tool-bars, scroll bars, and other sorts of controls. In addition to the external interface, there may be I/O processing routines, internal processing routines, and operating-system interface routines. The first step in component-level design is similar to that of any object-oriented design, in that we wish to identify the various distinct parts of a program, develop code for each part, and then integrate the parts. Unlike object-oriented design, component-level design focuses only on the large-scale aspects of the program. The various parts of the program will be implemented as independent components connected with glue logic. The small-scale parts of the program will be embedded in the various components, and possibly in

the glue logic. The small-scale parts of the program will be developed using conventional object-oriented techniques.

As one divides the program into its various parts, one must remember that there are many third-party components available in the marketplace. To speed development, it is necessary to implement as much of the program as possible using these third-party components. In some cases third-party components and glue logic will be sufficient, however in most cases, it will be necessary to develop a number of custom components. It is sometimes possible to avoid custom control development by implementing features directly in the glue logic, but except in simple cases, this approach should be avoided.

The first step is to separate program features into a few broad categories, as has been done in Figure 4.1. In this example, existing components are used to create the user interface, the operating system interface, and the database interface. The specialized parts of the program, those that are too complicated to be implemented in glue logic, will be implemented as custom components. Although this sort of break-down is typical of many applications, there are many others where the breakdown will be somewhat different, and there may be cases where a significant number of the specialized parts of a program will be implemented using existing components.

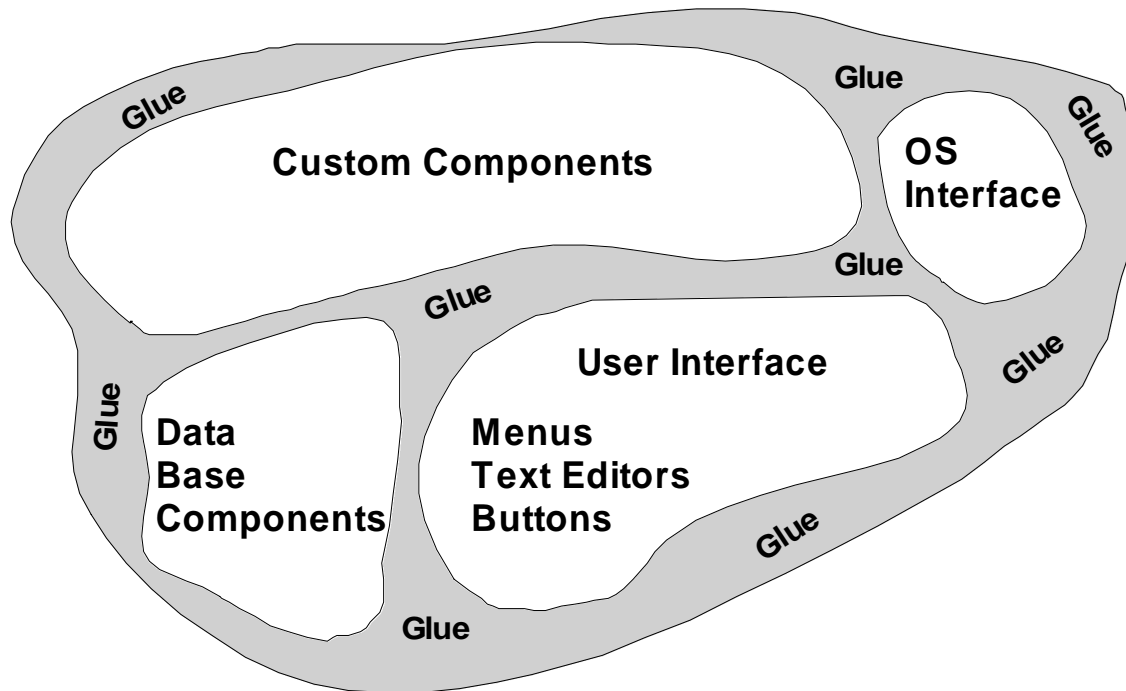


Figure 4.1. A Typical Component-Based Design.

4.3. Component-Based Development

Several skills are necessary for effective component-based design. One must have an extensive knowledge of available third-party components. One must have an idea of the typical sorts of services that are provided by third-party components, and how one typically interacts with them. One must also have an extensive knowledge of the component development process. To create custom components, one must know how the internals of a component are constructed and one must know how to create an appropriate interface for interaction between the component and its environment. Finally, one must know how to subdivide an application into components. (At least the portion that will be implemented in custom components.) One must have a good feel for which features should be combined into a single component, and which features should be implemented as separate components. The most difficult skill is the ability to design and build a wide

variety of different components, and a major portion of this book is devoted to this topic. The remainder of this chapter will describe the process of breaking an application into components.

4.4. A Design Example

To illustrate the process of breaking an application into components, it is best to start with a specific example. To that end, we will start with the requirements for a reasonably complex application, and create a component-level design for it. The application in question will be contact-management program. We intend for this to be a specialized product used by a single company. The application will allow the salespeople to list and manage their prospects. We will assume that this application will be a replacement for an existing application, and thus must be able to read and process the files used by the existing application.

4.4.1. Application Requirements

The existing application uses a collection of text files, one for each prospect list. Each prospect is a person with a name, address, phone number, and email address. The prospect files have a rigid format, but are ordinary text files in every other respect. These files have been integrated into other software, so we don't have the option of switching to a database system, even though this would be the ideal solution. The program must be able to open several files simultaneously and integrate them dynamically. Contacts should be displayed in a grid format as shown in Figure 4.2. Eventually, it will be necessary to add horizontal and vertical scroll bars to this display.

Last Name	First Name	Phone	E-Mail	Address
Jones	Harry	333-4444	xyz@abc.com	6262 Des
Smith	Helen	719-2391	qed@done.com	1745 Cap
Johnson	Alice	123-4567	eye@ear.com	316 Story
Brown	John	222-3333	ear@eye.com	1732 E 12
White	Allen	555-1212	kxl@ibn.com	1237 East
Fisher	Mary	336-2111	Mary@Fish.com	316 Arlen
Carter	Jimmy	525-3132	Prez@Georgia.com	1225 Harl
King	Larry	211-1111	king@martin.com	404 S 4th

Figure 4.2. A Sample Grid.

The grid of Figure 4.2 will be the basic means for editing, deleting and adding entries. The grid will also be used to manage prospects. This will be done by marking several lines for current attention, and then executing a command to create a “current attention” list. The grid will be embedded in a larger window that provides menus and tool bars, resulting in a display that looks similar to that of Figure 4.3. (Remember that this is a preliminary drawing, not a screen-shot of the final product.)

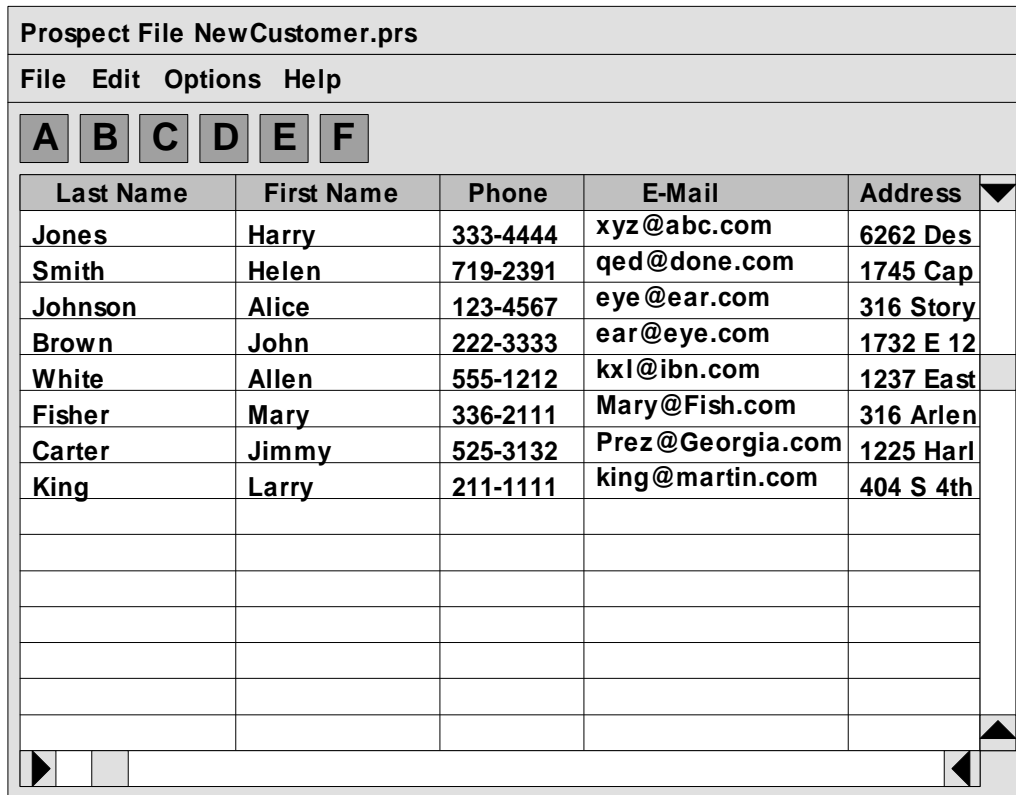


Figure 4.3. Main Window Design.

Although it is not necessary for us to specify every detail of the application at this point, it is useful to focus on a few low-level technical details. To that end, we will discuss the placing and managing of phone calls. The user must be able to tag entries for current attention, and must be able to save multiple current attention lists. The tool bar (or menu) will be used to choose between editing and selecting names for current attention lists. The user must be able to process a current attention list by calling everyone on the list. (The tool bar will be used to invoke this function.) A special window similar to that pictured in Figure 4.4 will be used for processing current attention lists. This window will enable the user to select a specific prospect and call that person by clicking a button labeled “call.” This task will be performed on an office computer equipped with a speaker-phone modem.

Current Attention List: Temporary		
Last Name	First Name	Phone
Jones	Harry	333-4444
Smith	Helen	719-2391
Carter	Jimmy	525-3132

CALL **HANG UP**

Figure 4.4. Current-Attention-List Window.

Internally, a call involves several steps. First the program must dial the person's number using the computer's modem. Once the call connects, the modem's speaker-phone function must be activated so the sales person can talk to the prospect. If the call doesn't connect after four rings, or gets a busy signal, the program must hang up and pop up a special window that enables the sales person to send E-Mail to the prospect. As Figure 4.5 shows, the E-Mail window will have a "send" button that sends the E-Mail and closes the window.

E-Mail to Jimmy.Carter@georgia.com
<p>Jimmy:</p> <p>I tried to call you on Thursday, but your phone was busy. I'll call back after the election.</p> <p>Jean Duprez</p>
SEND

Figure 4.5. E-Mail Window.

4.4.2. Application Design

A preliminary division of this program into components is relatively easy. Although it will be necessary to do a more detailed job before we are finished, a preliminary inspection of the requirements would produce a list similar to that pictured in Figure 4.6. Along with the name of each component (or instance of a component) we have indicated the source of each part.

Item	Source
Three windows with frames.	Development Language (VB, Delphi, etc.)
The menu in the main window	Third party component
The tool bar in the main window	Third party component
The grid display in the main window.	Third party component
The grid display in the call window	Same as previous
The text editor in the E-Mail Window	Built-in editor control
A file handler for reading and writing files	<i>Custom component</i>
A modem interface for placing calls, detecting rings, busies, and pick-ups.	Third Party Component
An E-Mail interface for sending E-Mail	Third Party Component
A configuration management tool for keeping track of persistent properties (maybe).	<i>Custom Component</i>
Buttons and scroll bars.	Built-in components
An internal file manager for maintaining open files.	<i>Custom Component</i>

Figure 4.6. Component-Level Program Breakdown.

The breakdown of Figure 4.6 assumes that we have spent some time studying the available third party controls, and have a good knowledge of what is available. (Such knowledge can be obtained by going through the catalogs of a few component clearing houses. We won't concentrate on that aspect of things here.) We can use the built in controls provided by our development system to provide windows, buttons, scroll-bars, and text editors.

A major part of the program will be provided by third party components. The entire user interface consists of third party components and built in features of the development language. The operating system interface to E-Mail functions and to the modem will also be handled by third party components. Much of what appears to be the functionality of the program will be implemented in glue logic. Specifically the code implementing things like placing calls will be implemented as event handlers for buttons and the various other components of the program. For example, the CALL button of the current attention window might have the Visual Basic code given in Figure 4.7.

```
Sub CallButton_Click( )
    Dim Number As String
    ' Get Number from selected row
    Number = DataGrid.FieldOfSelectedRow(2)
    Status = Dialing      ' global with defined variable
    RingCount = 0        ' Number of rings
    ModemComp.Dial Number
End Sub
```

Figure 4.7. Call Button Handler.

The code of Figure 4.7 makes several assumptions about other components. It assumes that there is a array-property in the grid component that permits access to the various fields in the row. (Some mechanism to do this must surely be available.) It also assumes that the modem component will handle the dialing function. Let us assume that this is the case, and that the response from the modem is in the form of several different events, RING, BUSY, CONNECT, and ERROR. The code for the RING and CONNECT events is given in Figure 4.8.

```
Sub ModemComp_Ring( )
  ' Global variable initialized to zero at start of call
  RingCount = RingCount + 1
  If RingCount >= 4 Then
    ModemComp.HangUp
    Status = NoCall
    RingCount = 0
    Load EMailWindow
  End If
End Sub

Sub ModemComp_Connect( )
  ModemComp.ActivateSpeakerPhone
  Beep
End Sub
```

Figure 4.8. Ring and Connect Handlers.

The major problem in the development of the application is the creation of the customized file-management components. File management components are needed because text files must be loaded in their entirety and copied into the grid of Figure 4.2. The grid itself will probably not be suited to storage management. In fact, for some third party grids only the visible portion of the grid can be stored in the component. The grid will issue events to obtain data new for scrolling operations.

As files are modified, it will be necessary, at some point, to write the modified data back to the disk. There needs to be some mechanism for keeping track of whether a file has been modified, and querying the user for saves at appropriate times. As the requirements state, it must be possible to open several files at once and combine them in various ways. In addition, there needs to be a mechanism for handling current call lists.

The first step in creating our custom components is to create an internal representation for prospect files. We will use a linked list for the individual prospect entries, and a Header object to contain the linked list. The file name, and other global properties of the file will be contained in the Header object. Figure 4.9 illustrates the organization of these objects.

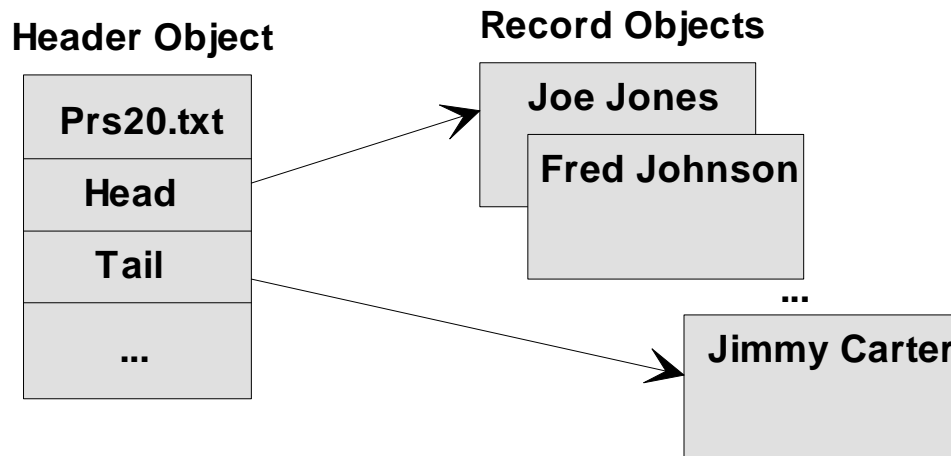


Figure 4.9. Internal Object Design.

The Header object of Figure 4.9 represents the internal form of a file, and it is proper to think of each Header object as a file. All access to the file must go through the Header object. The Record objects will be protected and available only through accessor and mutator functions. The Header object will provide all of the functionality required to manipulate files. There will be functions to access entries, delete entries, add entries, and replace entries. In addition, there will be functions for enumerating entries. Other functions will be used to sort entries into an appropriate order. There will be functions to combine lists in various ways, purge lists of duplicates, and perform other useful operations. The Header object is the heart and soul of our program, and its design will be completed before proceeding with the remainder of the application.

Once the Header object is designed and implemented, we can proceed with the design of the file handling components. The types of components that we will create, and the overall structure of the design may be somewhat surprising. To begin with, we will *not* create a single all-things-to-everyone component. Instead, we will create a family of

components, each one of which implements a single feature. The overall structure of the components is shown in Figure 4.10.

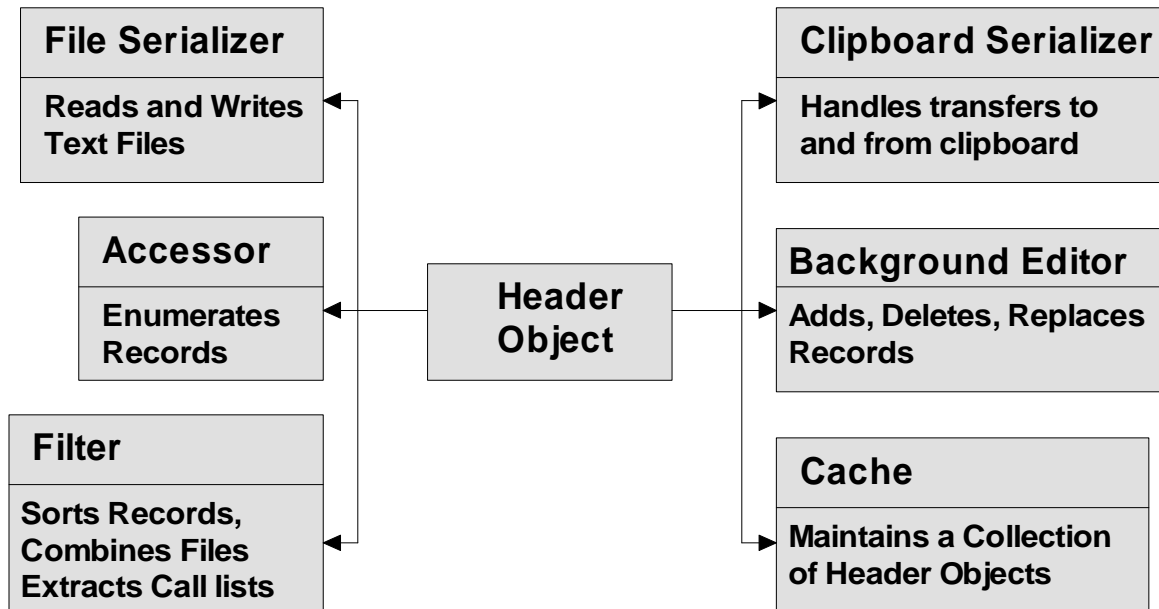


Figure 4.10. Custom Components.

The components of Figure 4.10 implement the data management functions of the application. The file serializer reads and writes text files. It will accept a file name and return a pointer to a Header object (or an error code). It will also accept a pointer to a Header object and write the object to a text file. Access to directories and file names will be provided by standard features of our development environment, such as common Open and Save-As dialog boxes.

The Accessor component gives the glue logic access to the Header object. It can be used to enumerate a range of records, or to give access to a specific record. The Background Editor will be used to modify a Header objects and the records within it. Both the Accessor and the Background Editor will have a pointer to the currently visible

file. (it may be advisable to combine these two components into a single component, but for the time being we will keep them separate.) The display will be managed through an interaction between the Accessor and the grid component of the main window. Changes to records will be managed through an interaction between the grid component of the main window and the Background Editor component. To make these interactions work, the grid component must issue Change events whenever data items change value. The grid must also issue Scroll events to obtain new data for the display.

The Cache component will be used to maintain the collection of files that are open but not visible. The Cache will be able to enumerate the names of the files it contains for display on menus. The Filter component will be used to perform major modifications on open files, such as sorting records or combining two lists. It is possible to implement each different type of operation in a separate filter. Access to clipboard cut, paste and copy operations will be provided by the Clipboard Serializer.

As mentioned above, we have created this set of components by implementing each separate feature as a separate component. There are good reasons for doing this. When two features are separated into different components, the coupling between the two features is minimized. The only interaction between the features is through a rigidly defined interface which cannot be subverted. The six components of Figure 4.10 could easily be written by six different individuals, thus maximizing available human resources for the project. Modifying one feature will have minimal impact on other features, because the features are isolated from one another. Adding new features can be done by adding new components, which will also have a minimal impact on existing features. For example, we could add a database feature to our application by adding a database

component. Such a component would read a prospect list from a database table and convert it into a Header object. Once created this object could be handled exactly like any other Header object. When the user is finished with the object, it could be written back to the database by the same database component.

Although we have said little about the current attention list we have not really ignored the issue. If we assume that the current attention list and the prospect list have the same internal format, we can handle current attention lists without creating any new components. Note that we are assuming that the filter component is capable of copying selected records from an existing Header object into a newly created Header object. This will create a new current attention list. A second instance of the Accessor component can be used to serve data for the current attention call window, thus getting two pieces of functionality for the price of one.

It may seem that our technique of subdividing an application into a set of independent features, each of which has minimal functionality, will eventually lead to an essentially unlimited variety of components. Surprisingly this is not the case. In a broad sense, there are only a very few types of components. Consider serializers, for example. Although every serializer does something slightly different, all serializers are pretty much the same. Their interfaces are all roughly identical, they perform similar functions, and the principles of their construction are roughly the same. The same thing can be said of accessors, caches, filters, and background editors. If we categorize components at this level, we will end up with a list of a dozen or so categories. Each category has its own well-defined design methodology, and each provides a specific type of functionality for an application.

Component categorization is an enormously powerful tool for designing component-based applications. One can go through a set of requirements and assign the various features of the application to glue logic, third party components and custom components. Each part of the program that is to be implemented as a custom component can be assigned to a particular component category. If a feature doesn't fit any of the categories, it probably needs to be divided further. When all features have been assigned to one category or another, development can proceed through a set of well-defined design methodologies, one for each category.

Although it is usually desirable to subdivide application features as much as possible, there are cases where different features should be combined into a single component. Third party components are one such case. When creating a component for the third party market, it is important to provide as much functionality as possible to make the component more saleable and easier to use. Another case where combining features is useful is when an excessive number of custom components is required for a single application. If you have an application that has hundreds of custom components, you should probably think about combining several features into a single component.

4.5. Component Categorization

In today's marketplace, there is a bewildering variety of third party components, a variety that almost defies description. Most attempts to categorize components have focused on applications. There are web-access components, word-processing components, telephony components, personal information manager components, spreadsheet components, and many, many others. From a component-user's point of view this sort of categorization makes sense, because users are generally looking for a specific

set of features for a particular application. A component-developer, however, needs to look at things differently. Is there really any difference between a text-editor that is used for word processing and one that is used in a personal information manager? No, not really. There may be some minor differences in features, but the design-principles of the two editors will be the same. Since we are focusing on the design of new components, we will adopt a view that is based on internal design principles rather than end use. Viewed from this perspective, there are remarkably few component-categories. To simplify things further, we will break our categories into three broad classes, *Visible Components*, *Invisible Components*, and *Other*.

4.6. The Visible Components

Visible components are those that normally have a run-time view. Drawing the run-time view is a significant part of the design. In most cases, the end-user will be able to interact with the component display using either the keyboard or the mouse, or both. The categories included in this classification are *Models*, *Editors*, *Displays*, *UI Widgets*, and *Decorations*. Although components in these categories normally have run-time displays, it is possible to implement them without one, and it is possible to use them with the run-time display turned off.

4.6.1. Models

A *Model* is a component that acts as a wrapper for an object-oriented model of some real-world object. The usual objective in creating a Model is to perform a simulation of some sort. The Model can be used to create games like Parcheesi, Chess, or Tic-Tac-Toe. Central to the Model component is an internal object-model that represents the real-world

object encapsulated by the model. The visible display provides a visible rendering of the internal object-model. Usually, the end-user is capable of interacting with the visible display, either through mouse-clicks or keyboard actions. In most cases, the user interaction will be in the form of mouse-clicks that are passed to the container as events with informational parameters.

Events are used in two ways, the first is to pass information about user interaction to the component's container, the second is to report exceptional conditions to the component's container. The user winning the game is an example of an exceptional condition.

Properties are used in three ways, as rendering parameters, as model specification parameters, and as model informational parameters. For example, in the Tic-Tac-Toe model, a property could be used to indicate the size of each square. Since this property does not affect the internal object-model, it is a *rendering parameter*. Another property could be used to indicate the number of horizontal and vertical squares in the display (going beyond the traditional 3x3). In reality, this property actually affects only the internal object-model; its affect on the display is incidental. Therefore it is a *model specification parameter*. We could use a third property to indicate whether the current state of the model represents a win for one side or the other. The value of this property derives from the current state of the model; its value cannot be changed without changing the model. Therefore it is a *model informational parameter*. When either the model state or the visible rendering is complex, methods can be used instead of properties to implement these three types of functions.

The primary use for *methods* is as object-model manipulator functions. In the Tic-Tac-Toe model, a method could be used to insert an X or and O in an empty square, or to request that the computer make a move.

In the design of a model, we generally recommend a *passive object* design philosophy. In this design philosophy, the component responds *only* to property changes and method calls. User actions are reported to the container as events, but no action is taken in response to them. Responding to user actions is the responsibility of the container, which can elect to call a method, or set a property value in response to the event.

Variations on the Model are the Invisible model, and the Multi-Object Model. Invisible models are those that require no user interaction or visible display, while multi-object models are those that can obtain their internal object models from some external source.

4.6.2. Editors

An *Editor* has some elements in common with a Model. Like a Model, it has an internal object-model. Like a Model, it has a visible display. However, unlike the Model, the Editor is used to create the internal object-model and to perform extensive modifications on it. Editors are inherently multi-object in nature. They must be able to save and restore their internal object models. In most cases, persistent storage will be used for this purpose, but semi-persistent objects can also be used for this purpose. (A semi-persistent object is an object whose lifetime is independent of its creator. See Chapter 19 for more details.) The most important distinction between Models and Editors is the response of the component to end-user actions. In general, an Editor will perform

complex actions on its internal state in response to user input. Little or no action will be required on the part of the container.

One of the most important aspects of Editor design is the design of the internal object-model. Generally, the internal object-model, as represented inside the component, will differ extensively from the data that the editor stores on persistent storage. For example, a text-editor that uses simple text files for external storage may have a complex internal model to facilitate adding and deleting lines, scrolling, searching, and other required operations. Chapter 7 covers object-model design in detail.

Events are used for the monitoring and confirmation of user actions. For example a **Change** event could be used to signal a user-initiated change in the internal object-model. A **Delete** event could be used to alert the container to a pending delete action, and request confirmation of the action. An **OKToSave** event could be used to signal the pending erasure of unsaved changes, either by erasure of the existing object-model, or by a pending load of a new object-model. (These two actions correspond to the “New” and “Open” commands in a file editor.) Events can also be used to monitor the progress of complex changes, such as a “Replace All” command. An **OKToChange** event could be used to confirm each individual change.

Properties are used to provide informational parameters about the display and the internal object-model. For example a **LineCount** property could be used to give the number of text-lines in the current object-model, while a **FirstVisible** property could be used to indicate the first line visible in the display.

Methods and properties are used to manipulate the state of the internal object-model and the state of the display. The **FirstVisible** property mentioned above could also be

used to scroll the display to a specific point. A **ScrollTo** method could be used for the same purpose. The general rule about state manipulation is that any operation that can be performed by the user must also be available to the container through properties and methods. If the user can select text, then the container must be able to select text. If the user can draw a new rectangle, then the container must be able to draw a rectangle. There is no consensus about whether properties or methods are best for such state manipulations, but the tendency is to use methods for this purpose because methods generally permit a wider range of parameters to be supplied to the operation.

Methods are also used to perform menu-based operations on the internal object model. In particular, the clipboard operations, Copy, Cut and Paste, are traditionally implemented as menu selections on an Edit menu. Since components generally do not have their own menus, methods are used to transmit these menu commands to the component.

It is at least theoretically possible to create an Editor component that has no display or any end-user interaction. For such a component, all editing must be done through the programmatic interface. An example of such a component might be one that permits an HTML document to be created programmatically without external user interaction.

4.6.3. Displays

A Display component is used to make information visible to the user without providing any editing capability. One important use of Display components is to display information from proprietary file formats that the user is not permitted to edit. Some commercial examples are the Acrobat Reader control, and the PowerPoint viewer control. These components are distributed free of charge. The corresponding editors are

commercial products that must be licensed for a fee. In a sense, this type of Display is a restricted form of an editor. The features of the Display component are generally a subset of those provided by the full product. Because the features of such a Display component are adequately covered in the Chapter 7 on Editors, we will not present a separate discussion of this type of Display component.

A second, and more interesting type of Display component is one that is used for the *Visualization* of data. Visualization is a complex research area in its own right, and as such is beyond the scope of this book. However, we will present some simple types of Displays that permit visualization of simple objects. Like Models and Editors, Displays have an internal object-model. Depending on the complexity of the component, the internal model can be something as simple as an array of numbers, or something as complex as a relief map of the United States.

Events are generally used to report exceptional conditions, but can also permit some user interaction with the display. If user interaction is permitted, it is generally limited to mouse-clicks, which are passed to the container as events, sometimes with informational parameters.

Properties are used to provide informational parameters about the internal object model. For example, a Display component that is used to present a bar-graph could provide a property called **LargestBar** that gives the height of the tallest bar. Properties can also be used to modify the display parameters, such as zoom factor, and background color, as well as other parameters that are specific to the type of the Display.

Methods are used to supply data to the component. If the data is simple enough, properties can be used for this purpose.

By its very nature, a Display must always have a visible display.

4.6.4. UI Widgets

UI (User Interface) Widgets are the first, and still the most common type of component. There is much discussion about UI Widgets already available. We will repeat much of this information here, but have little additional information to add to it. The only difference in our point of view is that we do not consider data-entry components to be UI Widgets. We prefer to view such components as Editors, because they are substantially different from other types of UI Widgets. In our view, the main purpose of UI Widgets is to translate mouse-clicks into more useful types of events.

Although some UI Widgets have an internal object-model, many do not. A drop-down list has an internal object-model that gives the content of each list item, while a button has no internal object-model. (This is an arguable point. Some may consider the caption to be part of a button's internal object model.)

Events are used to report user actions to the container. In some cases, events can also be used for exception reporting.

Properties are used as display parameters to control the size and colors of the display. They can also be used to manipulate the display in limited ways.

Methods are used to supply data for the internal object-model, if any.

Surprisingly, there are examples of invisible UI Widgets. Technically speaking, these components are *transparent* rather than invisible, but they are generally referred to as invisible components. They are used to provide user interaction with objects that do not normally permit such actions. They are typically drawn on top of another object to make that object, or some portion of it, sensitive to mouse clicks.

4.6.5. Decorations

Decorations are UI Widgets that do not permit user interaction. They are used only to enhance the quality of a display. The most common examples are labels and geometric shapes, but there are more complex types of Decorations. For example, BeCubed Software markets a component they call *MhMarque*, which displays a scrolling text banner. A Decoration component does not have to be purely decorative. It can transmit limited amounts of useful information, but it is far less complex than a Display component, or a UI Widget.

Events are generally not used. *Properties* are used to change the properties of the display. *Methods* are generally not required.

4.7. The Invisible Components

The class of Invisible Components includes those components that do not generally have a visible display or any user interaction. In some component technologies, such as ActiveX, the component will be visible as an icon at design time, but invisible at run time. In other technologies such as COM, the component is never visible. Because component-level programming has its roots in visual programming, most people tend to ignore the invisible components. However, the class of invisible components is an emerging area that provides a number of interesting and useful component categories. In some cases, the functions of an invisible component can be combined with those of a visible component. In some cases it may even be *preferable* to do so. However, because each category of component can be implemented as a separate entity, we prefer to treat each category separately.

The main categories of invisible components are *Filters*, *Accessors*, *Function Libraries*, *Caches*, and *Serializers*. Because virtually any non-interactive program could be implemented as an invisible component, it is quite difficult to make this list of categories comprehensive. We believe that this list is relatively complete, but we admit to the possibility of some glaring omissions.

4.7.1. Filters

A Filter is a component that transforms one object into an object of a different type, or into a “better” object of the same type. The objects in question can be files or internal objects. There is virtually no restriction on the types of objects that can be used. If the component operates on internal objects, it is necessary to establish some mechanism for transmitting objects between different components, and for establishing ownership of the objects.

Because many traditional programs can be viewed as filters, it is tempting to simply take a traditional program and place it into a component wrapper and call it a filter. For example, a compiler can be viewed as a program that transforms source files into executable modules, so it may be tempting to convert an existing compiler into a component by embedding the existing implementation in a Filter component wrapper. This is probably not the best way to do it. Because components operate in a different sort of environment from traditional programs, it is not necessary or advisable to consider a *file* to be the only possible output of a program. (By the term *file* we also mean console and printer output.) In some cases we might want to consider a linked list, or some other complex object as the output of our Filter component. This permits applications to be broken down into components in some non-traditional ways. Returning to the example of

a compiler, we might wish to break it into several components such as, *parser*, *pre-processor*, *code-generator*, *optimizer*, *assembler*, and *linker*. Given sufficient internal memory, there is no need to use files for intermediate storage. We could simply pass complex internal objects from one component to the next. This not only eliminates the need for reading and writing intermediate objects, but also allows us to pass input data in a form that is readily accessible to the next stage.

There are two broad categories of filters, *hot* filters and *cold* filters. A cold filter is frozen. Given the same input it always produces the same output. A hot filter is configurable. Its output depends not only on its input, but also on its internal state.

From a component-design point of view, the most important part of a Filter is the programming interface, including the design of the input and output objects. (The design of the internal algorithms is also important, but it spans the range of computer applications, and is, for the most part, beyond the scope of this book.) *Properties* and *methods* are used to transmit objects to the Filter, and to retrieve output objects. *Events* are used only to report exception conditions.

In Chapter 11 we give several examples of filters.

4.7.2. Accessors

An Accessor is a tool that provides programmatic access, at the container level, to a complex object. The objects in question are generally difficult or impossible to access in the container's programming technology. An example of an Accessor component, is an HTML component that allows the user to access the tags and content of an HTML file in some reasonable way. Accessor facilities can be added to Model and Editor components.

Methods and *properties* are used to supply the internal object-model. Properties are also used to access various parts of the object model, while methods are generally used as iterator functions. Normally a set of properties will be used to access a sub-object within the object model, and the iterator functions will be used to determine which sub-object is currently being accessed by these properties. Accessor components generally have large numbers of properties and methods. Events are used to report errors in the object model and exception conditions in the iteration process such as **EndOfList** or **NoCurrentObject**.

4.7.3. Function Libraries

Because existing technology provides many different methods for creating and accessing function libraries, the use of Function-Library components is strictly optional. The main reason for embedding a function library in a component is convenience. A component is generally self-defining, so header files, and function prototypes are not required. Function libraries are especially useful in web-based applications, where they can be used to embed useful functions in a web page.

Function libraries have no internal state, so properties are not required. Function calls are implemented as methods. If desired, events can be used to report exception conditions, but more traditional methods of raising exceptions can also be used. If efficiency is a concern, a standard function library should be used rather than component Function Library.

4.7.4. Caches

A Cache is a tool used at the container level to store and organize data. It can be used to implement traditional data structures such as stacks, queues, and binary search trees. Access is generally done strictly through the properties, but Accessor functionality can also be added. A drawback of this type of component is that it is generally capable of handling only one type of data. If one needs both a queue of integers and a queue of real numbers, it is generally necessary to implement two different Cache components. Events are used to signal exception conditions, such as **EndOfData**.

4.7.5. Serializers

A serializer is a component that accepts semi-persistent objects from some source, serializes them and disposes of them in some fashion, usually by writing them to disk. The serializer also performs the reverse function of retrieving a serialized object and creating a new semi-persistent object from it. A serializer is the bridge between a file-format and a semi-persistent object.

In addition to reading and writing objects to disk, a serializer may write objects to the clipboard, or retrieve them from the clipboard. Since clipboard objects may pass from one address space to another, objects passed through the clipboard must be serialized. A serializer may also interface with an object storage and retrieval system such as Corba.

Editors are the most common examples of components that require serialization, but there are many other types of components that could be enhanced with serialization of their internal object models. For most existing components, the serialization function is built-in to the components that require it, however there are strong reasons for separating the serialization function from other components. The first reason is encapsulation.

Serialization is fundamentally different from the other functions performed by a component. It is so different, that it makes sense to have a separate component to perform this function. The second reason is file-format independence. The use of semi-persistent objects frees a component from dependency on a particular file format. Incorporating the serialization function into the component reestablishes this dependency. By separating serialization from other functions, we maintain file-format independence. By encapsulating the serialization functions into a separate component we also simplify the task of adding new file-formats to an application.

4.8. Other Types

The types of components described in this section are neither inherently visible nor inherently invisible. They tend to provide functions that are outside the “main stream” of application development (although in some cases the functions they provide are absolutely necessary), or they provide functions that are difficult to classify.

4.8.1. Service Wrappers

The purpose of a service wrapper is to provide simplified access to operating system services. Many of these components are invisible at run time, but some of them have a visible user interface. The quintessential example of this type of component is the Microsoft Common Dialogs control. Many of the dialog boxes that appear in programs for the Windows operating system are standard dialog boxes that are provided by an operating system program library. The two most commonly used of these are the Open File dialog and the Save As dialog. Although most these dialogs are accessible in Visual Basic and other container applications, they are somewhat difficult to use. The Microsoft

Common Dialogs component provides a wrapper for these dialogs. To display the Open File dialog, the programmer includes a Common Dialogs control in his or her program, and then calls the *ShowOpen* method of this control. The control itself is invisible at run time. Components in this category include wrappers for telecommunications, wrappers TCP/IP communications, multi-media players, and many others.

4.8.2. Containers

A container is a component that is used to hold other components. Any component can be designed as a container, and component design tools can provide the container functions automatically. It is conceivable that one could design an entire hierarchy of components that are contained within one another, but this has not yet proven useful in practice. A container can be used as a layout tool, permitting groups of components to be moved or sized as a unit.

One of the most useful types of containers is the tab control that provides several pages of components and permits the user to switch from page to page by clicking on a tab. Other containers provide run-time resizable windows and splitter bars that can be used to change the relative sizes of portions of a program's window.

4.8.3. Miscellaneous

In any classification scheme that purports to include all types of components, there will inevitably be a few types of components that don't fit well into the classification scheme. The reader is cautioned that the field of component level design is a highly volatile one, and something that is categorized as *Miscellaneous* today could become mainstream tomorrow.

One difficult-to-classify component is the Universal Data-Aware component. There are components that can be used to add database functionality to virtually any other component, including those that were not designed for such interaction. (See chapter 2, section 5.) Components that permit two other components to communicate with one another are exceedingly rare.

Also in this category are the more complex types of components that can be created using Java and other languages. These languages permit ordinary applications to be embedded in a component wrapper. This can result in a component that contains several other components interacting in a complex way. Such components tend to defy classification.

4.9. Conclusion

The design methodology of this chapter has four distinct phases. The first phase is dividing an application into third-party components, glue logic and custom components. This phase requires an extensive knowledge of existing third-party components, their functionality and their interfaces. The second phase is the design of shared objects that will be used by more than one custom component. The third phase is the specification of specific custom components for all application features that are to be implemented in this fashion. The fourth phase is the design of the components themselves.

The classification of components is intended to be used as a design tool for designing the custom components of an application. It focuses on design methodology rather than end use, and provides a specific design methodology for each category of component. Using this methodology, the design of a component-level application is a straightforward process that can produce reliable, full featured applications. In the remainder of the book,

we will look at formal design techniques for specific components, and provide a specific design methodology for each category of component.

4.10. Exercises

1. Examine your favorite MP3 music player. Suggest a component-level design for it.
2. Go to the EKS website (<http://www.kaser.com>) and choose your favorite game. Suggest a component level design for it.
3. Examine your favorite web browser. Suggest a component-level design for it.
4. Choose a program you use regularly, other than a game. (Something like a word processor or a spread sheet program.) Suggest a component level design for it.
5. Go to a third-party component vendor's site (e.g. <http://www.vbxtras.com>) and categorize 25 of the components you find there according to the scheme presented in this chapter.