

Categories for Component Level Design

Peter M. Maurer
Department of Computer Science
Baylor University
P. O. Box 97356
Waco, TX 76798-7356

Abstract

Several research problems are discussed, including communication mechanisms and the interface between components and the glue logic used to tie them together. A technique for engineering new component-level applications is presented. This technique divides an application into a set of tightly coupled parts each one of which can be placed in a specific category. Each of these categories has its own design methodology, most of which rely heavily on object oriented design. Once the features of an application have been divided into categories, they can be implemented as off-the-shelf components, new components, or as glue logic. The categorization scheme gives guidelines for the placement of each feature. The categorization scheme relies on the ability to pass pointers complex objects between components. The dangers of such practices are discussed, and suggestions are made for future research that might help mitigate these dangers.

Index Terms: Component Level Design, Software Design Methodology, Object Oriented Design

1 Introduction

The past several years have witnessed an explosion in component level design. Visual languages such as Visual Basic and Delphi have become the preferred languages for new application development. Third-party sales of ActiveX and JavaBeans components has become a major industry. The use of components has slashed the time required to create new applications, and at the same time has allowed programmers to add a wide variety of complex features to their programs.

From the most abstract point of view, the main reason for the success of components is software reuse. Components permit software reuse to a degree that is virtually impossible by any other method. Component interfaces are rigidly defined, and their implementations are completely isolated from one another. The elements normally associated with complex programming packages have been virtually eliminated from the component interface. These include such things as header files, complex new data types, user generated initialization and termination procedures, and call-back functions. Components do not require header files. They must be self-identifying through a process called *introspection*. In many technologies, the component interface supports nothing beyond simple data types. Components must be self-initializing and self-terminating. They must be capable of creating multiple non-interfering instances of themselves upon demand, and they must be self-contained with no required callback functions.

All of this has led to an interface that is remarkably easy to use. In most cases, the interface is built around three paradigms, the property paradigm, the method paradigm, and the event paradigm. The property paradigm is modeled after object variables. One is able to use a component property as if it were a variable. Its value can be referenced in expressions, and its value can be changed using assignment statements. Properties are significantly more powerful than variables, however, because all accesses to the property are monitored by the component. The component can perform complex actions in response to these accesses.

The method paradigm is modeled after object function calls. Method calls are a convenience rather than a necessity, since all required functions could be performed in response to property accesses. In many technologies, the operands and return values of methods are limited to simple data types.

The Event paradigm is based on the callback function, with significant differences. The most important difference is that event-handling procedures are always optional. Event handlers are keyed to a particular instance of a component, and not shared between instances. The component always operates as if all event handlers have been implemented, and is unaware of which handlers are implemented and which are not. The component does not call the event handlers directly, but instead directs its events through a generic event passing mechanism. Those events that have no handlers are discarded.

Despite the deficiencies of this interface (the limitation to simple data types and the lack of callback functions), it has been used effectively to create a stunning array of components. Components range in complexity from simple buttons to spreadsheets, word processors, web browsers, and image processors. Virtually any application that one can imagine is available as a component. What is more, these components can be integrated into a single application in a way that would be virtually impossible with a typical programming package. Using components, one could create an application containing a word processor, a spreadsheet, and a web browser all in the same window, and expect the whole thing to work. If one needed nothing more than the basic functionality of these components, one could easily create such an application in less than an hour.

To many researchers the pace of development in component level design has been overwhelming. New technologies seem to appear and disappear overnight. Innovations seem to focus almost exclusively on development tools rather than principles. It is not clear that component level development is governed by *any* underlying principles other than pure pragmatics. It is also not clear that if such underlying principles did exist that they would be of any value.

We are now at a point where it is appropriate to take a step back and look carefully at component level programming to try to determine its underlying principles. Component level design has been firmly established as a development technology, and has enjoyed a long enough success that it can no longer be ignored as a passing fad. The area of component level design is vast and largely unexplored at the theoretical level. The purpose of this paper is to make a beginning.

2 Understanding the Discipline

The first step in developing a theoretical basis for component level design is understanding the nature of the discipline. Unlike early high-level programming languages, which had a basis in mathematics, component level programming is purely an

engineering discipline. That is not to say that there are no underlying mathematical principles. We are merely suggesting that we should begin by treating component level design as a problem-solving tool.

The quest for better problem solving tools, those that go beyond high-level languages, has been a long one. At one time it seemed reasonable that we would be able to discover a uniform hierarchy of programming languages, starting with assembly language, moving on to high level languages, and then moving on to even higher level languages, and so forth. Somewhat surprisingly, these higher level languages never appeared. Object oriented languages, while representing a significant improvement in programming technology, are not “the next level of abstraction” in programming. Object oriented languages use the same programming primitives as non-object oriented languages. They represent a paradigm shift, not a new level of abstraction.

In the search for the next level of abstraction, it is our opinion that computer scientists have inadvertently rediscovered an engineering principle that is well known in other engineering disciplines. Namely, that it is difficult or impossible to design with intermediate-level off-the-shelf parts. At the intermediate level, there are so many design choices that it is difficult to create generic parts to meet every possible situation. It is usually necessary to carefully design each intermediate part to work efficiently with other intermediate parts. An automobile engine is a collection of parts that were carefully designed to work efficiently with one another. It is not a collection of off-the-shelf pistons, valves, and cams. When we wish to upgrade our computer, we replace the entire CPU. We don’t add an additional pipeline to the existing CPU. (Even if such a thing were feasible, it wouldn’t be practical.)

By the same token, software components represent a significant leap beyond the primitives of high-level programming languages. They are not simply the next level of abstraction. So then, what is a component, exactly? We believe that a component should be viewed as a complete program. Given the proper operating environment, it must be capable of running on its own, independently of other programs. It may not serve any useful purpose until it is integrated with its environment or coupled with other components, but it still capable of running independently of other programs. One can, for example, create an application that consists of nothing but a button. The application will serve no useful purpose, but the button will exhibit behavior that is independent of its environment. It will depress when clicked, for example.

At first this viewpoint may be a bit hard to swallow. After all, all component technologies require an environment that provides services that go far beyond those provided by the operating system. What is not so immediately clear is that this is true for *all* programs. Virtually any program written for any operating system depends on other programs for support. Consider the simple UNIX “Hello World” program illustrated in Figure 1. This program makes use of the standard output file “stdout.” Like any other file, stdout must be opened before it can be used, and the operating system does not provide this service. This service is actually provided by the UNIX shell which is an add-on program, not part of the operating system. Without the services of the shell, the Hello World program would not function correctly.

```
void main()  
{
```

```
        printf("Hello World\n");
    }
```

Figure 1. A UNIX Hello World Program.

Visual Basic and other component containers provide the same sorts of services to components that the UNIX shell provides for UNIX programs. This analogy is surprisingly faithful. Both the UNIX shell and Visual Basic provide a separate programming language that can be used to facilitate communication between independent programs. Both the UNIX shell and Visual Basic provide built-in commands (in Visual Basic they are called built-in controls). Neither the UNIX shell nor Visual Basic is the only operating environment that provides services to programs. There are many different versions of the UNIX shell, and there are many different operating environments that support components.

The main difference between the UNIX shell and Visual Basic is the environment in which they operate. The UNIX shell operates in a batch environment where files are the primary method of communication between programs. (Message queues and shared memory are specialized communication tools.) Each program runs in its own address space. Each program is its own “prime mover.” It begins execution, performs its specific task and terminates.

Visual Basic operates in an event-driven environment. Programs are passive entities responding to events produced by the environment and by other programs. A program will normally run continuously until it is explicitly forced to shut down. It is common for many programs to share the same address space. If we were to ask ourselves what the UNIX shell would look like if it were adapted to run in an event-driven environment, the answer would probably be “something like Visual Basic.”

Rather than representing an entirely new technology, Visual Basic and other visual languages are the logical successors of the UNIX shell. Note that when we describe components as complete programs, we are making a distinction between programs and applications. A Visual Basic application is usually a combination of many components. The same thing is true for certain complex UNIX applications. The standard “cc” application, for example, consists of four programs, the C preprocessor, the C compiler, the assembler, and the linkage editor. Each of these is a separate UNIX program. They are capable of stand-alone operation, but are seldom used this way.

Because an event-driven environment is inherently more complicated than a batch environment it is clear that the communication mechanisms used by programs must be correspondingly more complicated. This brings us to the first of the research questions, namely, “What is the best model for component communication.” The properties/methods/events model will probably not change significantly, but there are several obvious ways in which it could be improved. Properties and methods permit the component’s environment to use variables and call functions embedded in the component, while events allow a component to call functions embedded in the environment. It should be possible for the component to access *variables* embedded in the environment as well. (The ActiveX technology provides such a facility, called “Ambient Properties,” but the interface to these properties is neither simple nor straightforward.)

The communication used in the properties/methods/events model is all between the component and its environment. What about direct component-to-component

communication? Why shouldn't it be possible for one component to directly access the properties and methods of another component? How would such communication be specified? (There are some visual programming tools that allow one to simulate direct component-to-component communication, but these tools simply write the environment code for you. Communication must still take place through the environment.)

Another research problem is the underlying mechanisms used to support properties, methods and events. There are many different solutions to this problem, none of which is completely satisfactory. While it may turn out that the existing solutions to this problem are the best possible, this seems unlikely.

Another important problem is language support for the component interface. In some languages, such as Visual Basic and Delphi, components are well integrated with the language, permitting the properties/methods/events interface to be used in a natural way. In other languages such as C++ and Java, support for components is minimal to non-existent. Existing high level languages need to be extended with features that provide *simple* mechanisms for instantiating components, accessing their properties and methods, and handling their events.

As important as these issues are, they do not address the more fundamental problem of application design. It is this problem to which we will devote the remainder of this paper.

3 Application design.

By application design, we mean creating a new component based application from a set of formal requirements. There are two approaches to this problem. The first is to base the entire application on existing components. The application will be created by a cadre of programmers who are experts at using components, but who are either unable or unwilling to design new components. A major part of the design process involves searching for and purchasing existing components. This process is already in wide use, it works well, and we have little to say on the subject.

The second approach is to use a cadre of programmers who are experts at using components, but who are also capable of creating new components should the occasion demand. It is this second scenario that presents the most challenging design problems. The development of the application will take place on two levels. A portion of the application will be implemented as a set of new and existing components, and a portion of the application will be implemented in the host language, Visual Basic, Delphi, or Java Script, or some other suitable language. (The host language coding is sometimes referred to as "glue logic." This term is useful, if somewhat misleading.)

The first problem that the application designer faces is determining which portions of an application should be implemented as components, and which portions should be implemented as glue logic. Once this determination is made, it is necessary to determine the component structure of the application. In particular, the designer must determine which features should be implemented as a single component, and which should be combined together into a single component. We wish to provide guidelines for all of these activities.

To help the application designer decide which features to implement as components we have created a set of component categories. This set of categories can be used to pinpoint those features that are readily implementable as components. We then will

provide a generic design methodology for each category. These design methodologies will allow the application designer determine the extent and usage of each component. We will also provide some guidelines on what features should be implemented in glue logic and how the glue logic should be structured to integrate components into an application.

Although our set of categories is small (we have identified only thirteen categories) it encompasses virtually every component that is available today. We have also identified at least one category (Serializers) that is not represented by existing components.

4 Component Categories

In our effort to develop a theory of component level design, we have made an extensive study of existing components [1-10]. We have also experimented with the component level design process by creating dozens of components and component-based applications [11]. This process has led us to identify several categories of components. All components in a particular category have a similar design methodology, and are integrated into an application in more or less the same way. It is important to note that we categorize components by how they are designed and by how they are integrated with other software. We do not categorize by end-use. We would consider a flow-chart editor and a VLSI layout editor to be in the same category (Graphical Editor), even though their end uses are quite different. It is also important to note that we are considering *all* aspects of application design, not just the design of user interfaces. There was indeed a time when the primary use for components was in user interface design, but the art of component level design has advanced far beyond this point. Component level design has now penetrated into virtually every aspect of application design, to include many applications (such as compilers) that are traditionally considered to be batch-oriented.

We have identified the following component categories which will be discussed in greater depth in the following sections: User Interface Widgets, Editors, Serializers, Models, Filters, Background Editors, Displays, Accessors, Caches, Decorations, Function Libraries, Service Wrappers, and Containers. Although this list includes virtually every component we have encountered, we make no claim that it is comprehensive.

4.1 User Interface Widgets

The most widely available component category is the User Interface (UI) Widget. The purpose of a UI Widget is to translate user actions into commands that must be executed by the application. A wide variety of UI Widgets is available, so much so that people sometimes mistakenly assume that *all* components are UI Widgets. The collection of existing UI Widgets is so extensive that application designers will seldom have to create new ones. However, if creation of a new UI Widget is necessary there are four steps that the designer should follow, the design of the internal state (if any), the design of the visible display, the design of the user interaction, and the design of the event structure. Examples of UI Widgets with internal states are toggle-buttons that remain down when clicked, and scroll bars with sliders. The component designer must provide properties and methods for setting and retrieving the internal state, and must also determine how the internal state is to be made visible to the user.

When designing the visible display, the designer must determine whether the component will be scalable or of fixed size. Scalable components are more versatile, but

fixed-sized components can be rendered more simply using bit-mapped graphics. If the UI Widget is to be a metaphor for some physical object, the designer must make sure that the display accurately represents the object. If the component state is represented by visible changes in the display, the designer must determine whether a simple redrawing of the component is sufficient or whether more sophisticated animation techniques must be used. (For most UI Widgets, simple redrawing will suffice.)

When designing the user interaction, the component designer must first select an input device that will trigger the interaction. In most cases this will be the system mouse, but it is possible to use other input devices. If the mouse is selected as the input device, the designer must determine if the interaction is to be simple or complex. A button represents the case of a complex user action. To trigger a command button, the user must press a mouse button and release it while the mouse cursor remains located on the visible display. No single user action triggers the button. Other components react immediately when a mouse button is pressed. When a complex interaction is used, a temporary internal state must be used to coordinate the elements of the user interaction. Finally, the designer must determine whether the component will give some visible indication that it has been triggered. This may require a timer-controlled momentary change in the visible display.

UI Widgets communicate with the rest of the application through events. A careful design of the event structure is necessary when the component issues more than one type of event. There are two fundamental approaches to issuing multiple events. One can create a single physical event with a type code to indicate the specific type of event that has occurred, or one could create a separate physical event for each logical event. In the first case the event-handling code for the component can be gathered in one place. This will make the code more accessible to maintenance engineers, but will also require that every event to be encoded and decoded as it is passed across the component interface. The use of multiple events is somewhat more efficient and permits selected events to be easily ignored.

4.2 Editors

An editor is a component that permits the user to create and modify some significant entity such as a text file or a graphical display. (Editors do not have to be file oriented.) They come in two distinct flavors, text editors and graphical editors. It is our belief that the third-party component market has so thoroughly explored text editing that there is little or nothing that the component designer can do to improve upon it. Existing text editors range from simple data entry boxes to complex word processors with multiple fonts and embedded graphics. A full range of data entry boxes is available, from simple text entry to formatted numeric input. Because of this, we will focus our attention on graphical editors.

The most significant problem in the design of a graphical editor is the design of the graphical container. The graphical container, which should be a single object, must support the creation and deletion of graphical objects, as well as selection, movement, scaling, and other parameter changes to graphical objects. Designing an effective organization technique for a collection of graphical objects is by no means a trivial problem. Entire research projects have been devoted to it. The problem becomes especially difficult when the drawings contain massive numbers of graphical objects, as

in VLSI design and CAD applications. However, we are not so concerned with the internal structure of the graphical container as we are with its external interface. As long as the object is capable of responding to the appropriate external requests, we are not concerned about its internal structure. Our view of the logical structure of this object is presented in Figure 2.

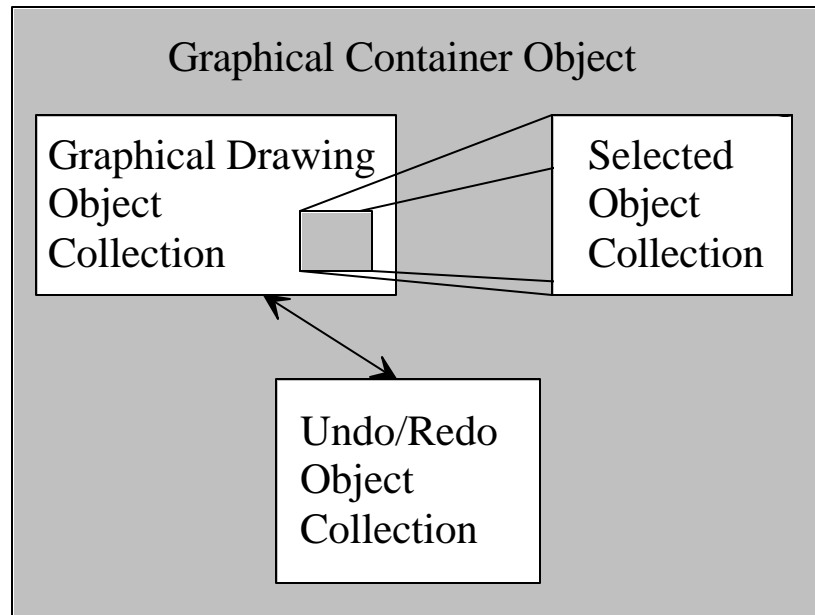


Figure 2. Graphical Container Structure.

As Figure 2 illustrates, we view the graphical container object as having three primary collections of objects. The first is the collection of the graphical objects themselves. The second identifies which of the graphical objects are selected, and the third implements the undo/redo facility (if any). The programming to support these three collections should be fully encapsulated within the graphical container object, possibly with several levels of inheritance and aggregation hierarchy.

After the design of the graphical objects and their containers, the most important aspect of editor design is the drawing routine. The drawing routine must be able to query the state of the graphical container and draw it, or portions of it, when required. If the graphical objects are designed using object oriented programming and polymorphic types, the drawing routine can be implemented as functions of the graphical objects themselves. (We recommend this approach.)

The user interface of a graphical editor must be *modal*. That is, certain user actions will perform different functions depending on the internal state of the component. For example, in one mode sweeping out a rectangle with the mouse will select all objects within the rectangle, and in another mode it will cause a new rectangle to be added to the drawing. A single *select* mode can be used for selecting, moving, and scaling graphical objects. Other general modes may permit rotation, shearing, and reshaping points. For drawing purposes, there should be one mode for each graphical object. This mode will permit objects of that type to be drawn.

The system mouse will be the primary method for the user to interact with the drawing, but the mouse operations will not be simple. Simple mouse operations must be

grouped into transactions. A mouse transaction is a button-press event followed by a (possibly empty) set of mouse movement events, and ending with a mouse button release event. To permit visible feedback during the mouse transaction, each transaction must be specifically geared to the current mode. Once a mouse transaction is complete, the information gathered from it is passed to the graphical container as a request. Mouse transactions should be independent of the graphical container.

It is not necessary to include an event interface with an editor, but there are circumstances under which such an interface can be useful. Events can be used to report changes to the drawing through a generic *Change* event. It is also possible to report the details of all user interactions through a more complex set of events. Events can be used to report exception conditions such as running out of memory or an attempt to delete a non-existent object.

In addition to mouse-related editing of objects, there are operations that take the form of commands. Two of these are changing the color of an object and deleting selected objects. The command interface should be implemented as a set of properties and methods. For example, if one wishes to change the color of all selected objects, one should be able to do so by assigning a new value to the *Color* property. If one wishes to delete the selected object, one should call the *Delete* method to do so. Direct user interaction with the component should not be used to trigger command-level functions. This design technique provides for the maximum flexibility in designing the user interface, and greatly simplifies the design of the component.

As a minimum, the command interface should support deletion of graphical objects, the undoing of changes, the modification of parameters that cannot be changed using mouse-based operations, and the creation of new drawings. We have deliberately omitted several important features from this list. These are saving and restoring drawings from files, cutting and pasting objects, and printing. We believe that these operations are distinct from the editing operations and can be separated from them by placing them in different components. However, to do this we require a component interface that supports more than just basic types. At a minimum we must be able to pass object pointers across the component interface. We currently do this by recasting pointers to integers, but there are certain risks associated with this practice. (See Section 5.)

Despite the risks associated with passing object pointers across the component interface, we highly recommend this approach.

4.3 Serializers

A serializer is a component with no visible display. (Such components are normally called *invisible* components.) The serializer performs a number of functions normally associated with the editing process. These functions include loading and saving files, performing cut and paste operations using the clipboard, and printing. Although most designers would prefer to integrate these features directly into an editor component, they are actually separate features unrelated to the editing process. There are several advantages to implementing these features as separate components. One important advantage is file independence. When files are handled by an independent component, the editor is independent of the file format used by the serializer. Several different serializers can be used to support multiple file formats, and new formats can be added

without changing the editor. The same could be said for printing. Printing options can be changed and enhanced without affecting the editor.

The most important advantage of using three separate components for editing, printing, and file operations is that the coupling between these three functions is reduced to a minimum. The only thing that the three features will have in common is the graphical container object, which can be passed by address between the three components. However, cutting and pasting, appears to be a different matter. Cut and paste operations must normally deal with the current selection, something that appears to be intimately associated with the editing process. These operations will alter the graphical container object by deleting graphical objects from the collection or adding graphical objects to it. The cut and paste operations will have an immediate effect on the display of the graphical objects, which is a function of the editor. It should be possible to undo cut and paste operations using the generic undo facility of the editor.

The main reason for separating cut and paste operations into a separate component is that the operations normally performed to copy data to and from the clipboard are virtually identical to those used for file I/O. When copying the current selection to the clipboard, graphical objects must be converted into serial form. Linked lists and other pointer-based structures must be converted into address-space-independent form. These operations are virtually identical to those required to write files to disk. While it is true that a cut and paste serializer will be more tightly coupled to the editor than file and print serializers, proper design of the graphical container object will tend to minimize this coupling. When designing the graphical editor, it is necessary to encapsulate the selection facility into the graphical container object as illustrated in Figure 2. This will enable the Cut and Paste serializer to query the graphical container to determine which objects are selected. Encapsulating the graphical object collection management into the graphical container object will allow the Cut and Paste component to modify the drawing. Encapsulating the management of the Undo/Redo facility into the graphical container object will permit cut and paste operations to be undone.

In addition to the design of the graphical container object, it is also necessary to carefully design the property/method/event structure of both the Editor and the Cut and Paste serializer. The Cut and Paste serializer should fire an event any time it makes a change to the graphical container object. The editor should provide a *redraw* method that permits the display of the graphical objects to be redrawn on demand. Finally, both components must provide properties that permit the address of the graphical container object to be passed between them. Figure 3 illustrates how a paste operation would proceed using these facilities. In this figure, it is assumed that the paste event is initially generated using a User Interface Widget that is physically unrelated to the Editor or the Serializer.

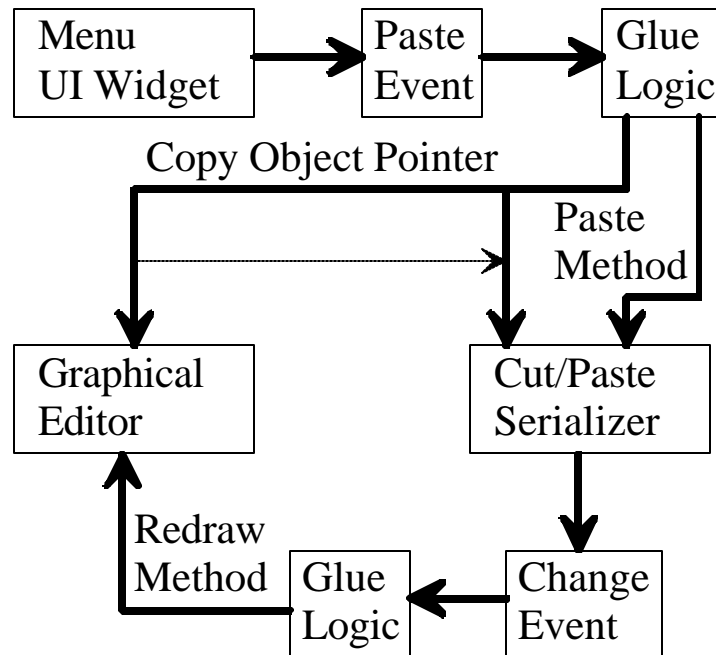


Figure 3. Pasting into a Graphical Editor.

4.4 Models

A model component is a wrapper for an object-oriented model of some kind. In most cases, the object-oriented model will represent something physical that has one or more visible representations. The model component differs from an editor in that the internal model is neither created nor substantially modified by the component. The object model is more or less frozen, with the component interface used only to modify the object-model parameters.

Models are interesting because they can be used to implement various sorts of interactive games and puzzles. The visible interface can be used for user interaction with the internal model. In most cases, user interactions with the visible interface will be passed through to the glue logic for processing. Consider, for example, the checkerboard illustrated in Figure 4. The internal model may provide nothing more than a visual display of the board and the pieces lying on it, with the details of the game itself contained in the glue logic. In such a scenario, user interaction with the board would be limited to mouse clicks. These clicks would be passed to the glue logic as events with parameters identifying the square clicked by the user.

A more sophisticated component may allow pieces to be dragged to new locations. In such a component, the drag operation would be used to determine the starting and ending squares for the move, but the move itself would be performed by the glue logic in response to an event issued by the component. If a complex strategy is used to perform computer moves, this strategy will generally be implemented inside the component for efficiency. Nevertheless, the strategy routines should be implemented as methods that are explicitly called by the glue logic rather than being invoked in response to a user interaction. This passive strategy for implementing complex actions simplifies the design of the component, and maximizes its versatility.

Formally, the design of a model component has four parts. The first, and most important is the design of the internal object model, which should be done using standard object-oriented design techniques. The second part is the design of the display. The issues are virtually the same as those of the display portion of an editor component. The third part is the design of the properties and methods that are used to manipulate the object model. In some cases, object model parameters and functions will be directly exposed using properties and methods, in other cases a rudimentary object model interface will be enhanced with more sophisticated user-oriented features. The fourth part is the design of the user interaction. In most cases, this will take the form of a collection of events that are fired in response to user interactions, but it may also include dragging various parts of the model with the mouse. Several examples of Model components can be found at [11].

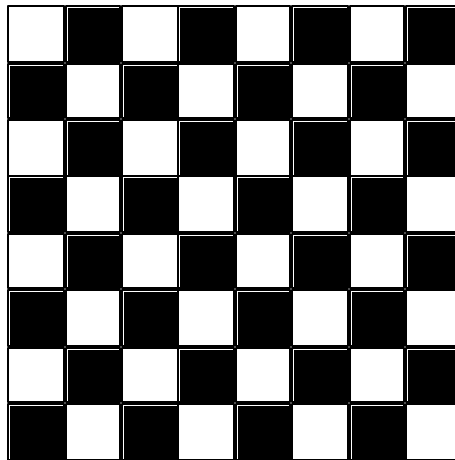


Figure 4. A Checkerboard Component.

4.5 Filters

A filter is an invisible component that implements a batch-oriented algorithm. It is the primary means for converting batch-oriented software into a component-level design. As is the case with most batch programs, the external interface is quite simple compared to that of a graphical component. Command-line parameters are supplied as properties of the component or as method parameters. The component may perform its function in response to a specific method call, or it may simply respond to an assignment to one of its properties. Figure 5 shows how the standard UNIX “cc” command might be implemented using filter components.

Figure 5 shows four filter components, the C Preprocessor, the C compiler, the Assembler, and the Linkage editor. Like their batch counterparts, these components communicate with one another using temporary files. The source file names as well as the temporary file names must be supplied using properties or method calls.

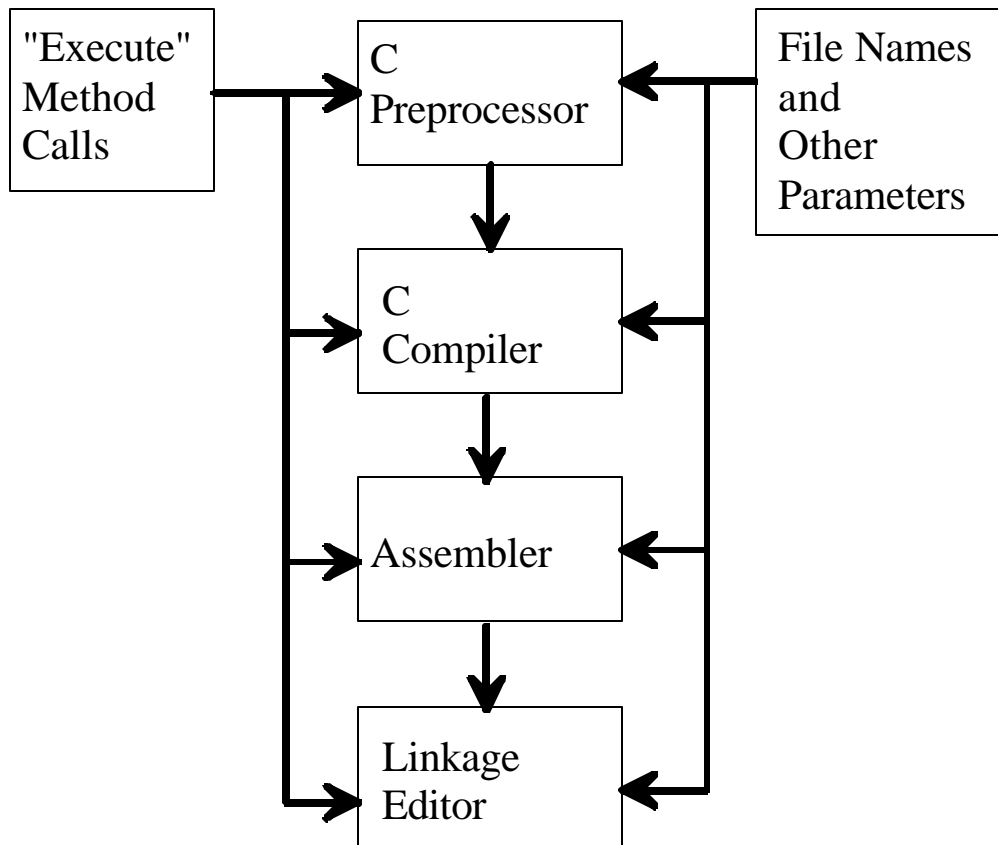


Figure 5. A Component-Level C Compiler.

Unlike their batch counterparts, Filter components need not communicate with one another using temporary files, but may also pass object pointers to one another. Considering the example of Figure 5, let us imagine for the moment that the first step in the assembly process is to translate the text output of the C compiler into an internal object with op-codes translated to internal codes, and labels replaced with pointers. Instead of creating an intermediate text file, the C compiler could generate its output directly in the assembler's internal format. The glue logic could then pass a pointer to this object from the C Compiler component to the Assembler component.

Although filters are used to implement batch functions, there is a hidden danger in directly converting batch programs into filter components. Because a batch program is designed to run as a separate process, and is designed to terminate once it has performed its function, the programmer can be careless about releasing unused memory without incurring a penalty. When the process terminates, all memory will automatically be released. In a component-based application, the functions implemented by the component may be invoked many times before the component terminates. In such a scenario it is necessary for the component to release all unused memory before terminating its function. When converting an existing batch program into a filter component, it is necessary to review the memory management structure to insure that memory is released when it is no longer needed.

4.6 Other Types

Compared to the types of components already discussed, the remaining types are of lesser importance, and we will discuss them only briefly. Two of these types, the background editor and the display component, are based on the editor component. A background editor component is an editor with no visible display. All editing operations must be performed through the property/method interface. In particular, properties and methods must be added to create new items, as well as to select, move and resize items. It is common to add background-editing facilities to an editor component.

The display component is an “editor component” with no editing facilities. It is capable of displaying information, but does not permit that information to be changed. In many cases a display component is created by reproducing an editor component and disabling its editing facilities. The most commonly encountered Display component is the Acrobat Reader component used to view PDF files in web pages.

An important subtype of the display component is the visualizer component. Such components are unrelated to editor components, and are used to create an easily interpreted visual display of complex data. Visualization is an important research area in its own right, and is much too complicated to be discussed here.

An accessor component provides programmatic access to a complex object through its property/method interface. It provides no visible display of the object. Access to the individual data items is usually provided through a set of properties, some of which may be arrays. Methods are generally used to iterate through collections of similar sub-objects. Events are seldom required, and are used only to report exception conditions. Accessor facilities may be added to editor, background editor, and display components.

A cache component is a component-level mechanism for the mass storage and retrieval of data. A cache may represent a simple storage object such as a stack or a queue, or something more complex like a database. The most commonly encountered example of a cache component is the Visual Basic data control. This component represents a database table or the result of a query and provides iteration facilities in addition to access to the individual data items. Apart from the Visual Basic data control and its clones, cache components are rarely encountered in the commercial world. It is our opinion that this type of component is severely under-utilized at the present time.

A decoration component is a UI widget with no user interface. Its sole purpose is to enhance the display of a graphical user interface. This is not to say that decorations provide no useful function. The most commonly encountered decoration is the text label, which is used to convey static information to the user.

The function library component is rarely encountered, and is of limited usefulness. It serves the same purpose as the more familiar static and dynamic function libraries that have been in use for many years. The most common use for a function library is for gathering information from the user. For example, a function library could be used to implement various design wizards for a graphical editor component.

Apart from the UI widget and the text editor, the service wrapper is the most commonly encountered type of commercial component. The service wrapper provides a component-level interface to an operating system facility. In most cases, these facilities are already available through other means, so we consider this type of component to be of minor importance. Nevertheless, service wrappers generally provide an interface that is far simpler and much easier to use than the native operating system service.

The main purpose of a container component is to serve as a host for other components. Container facilities can also be built into other types of components. The container is the primary means of supporting component hierarchies. At this point, it is not clear that hierarchy serves any useful purpose in component level design. Thus, at present, containers are of minor importance. It is our belief that as the science of component level design advances, we will discover that hierarchy is an *extremely* useful concept. When we reach this point it will probably be necessary to take another look at this category and perhaps replace it with several new categories.

Virtually every existing component fits into one (or more) of the categories we have described. However, component level design encompasses the entire scope of application software design, and it is likely that new categories will be discovered in the future.

5 Semi-Persistent Objects

Our scheme of subdividing applications and categorizing components depends heavily on the ability to pass complex objects by address from component to component. We refer to such objects as *semi-persistent* objects because they have a lifetime that is independent of the lifetime of their creator. By comparison, most objects are ephemeral, being created and destroyed during the lifetime of a single program. To be certain, this point of view is changing, particularly with object oriented databases [12] and remote object access systems such as CORBA [13]. However, these mechanisms support *persistent* objects that are written to magnetic disk or some other persistent medium. A semi-persistent object is never written to persistent storage even though it may exist for hours or even days by being passed from one program to another.

Unfortunately, existing systems do not support true semi-persistent objects. While it is possible to pass objects between components in some component technologies, in others one must resort to “benign trickery” to pass an object from one component to another. In the ActiveX technology, for example, it is necessary to recast pointers as long integers to pass them across the component interface.

There is some danger in passing objects from one component to another, especially when it is necessary to do so using a typecast. The most important problem is verification. In other words, when a component receives a pointer to an object of type X, what assurance does the component have that this pointer *really* points to an object of type X? Because components tend to be independently compiled programs, there is no way for a compiler to enforce type correctness. The component must depend on the good will and skill of the user. Furthermore, if a component mistrusts the source of the object pointer, what mechanisms can be used to verify the identity of the object? As yet, there are no fully satisfactory answers to these questions.

The problem is exacerbated by the use of virtual functions and polymorphic types. When a class defines virtual functions, all objects of that class contain pointers to the virtual functions. The reference is indirect through the object’s v-table, but it is a real reference nonetheless. Since a component is an independently compiled program, the module containing the component must contain the implementations of each virtual function. When an object is passed by reference to another component, these virtual function references are not retargeted to the new module, which must have its own copy of each virtual function. When a virtual function is called, the copy of the virtual function embedded in the object’s creator will be called, not the copy embedded in the object’s

current owner. If the object's creator terminates, the object becomes useless because its virtual function references have become dangling pointers.

When polymorphic types are used, this problem becomes even worse. Even if it were possible to retarget the virtual function references of an object to its new owner, a collection of polymorphic objects may contain objects that were not yet defined when the collection's new owner was compiled. The new owner will not have access to the virtual functions of the new object.

There are, of course, solutions to these problems, the most obvious of which is to provide a separate component (or dynamic function library) to manage the creation and destruction of semi-persistent objects. Such solutions have their own problems, which are too complicated to discuss here. We believe that the true solution to the problem lies in object management systems such as CORBA. We believe that remote object management should be an integral part of all modern operating systems. We believe that there should be integrated mechanisms for passing objects, both data and functions, between different programs, different address spaces, different operating systems and different hardware. We believe that there should be mechanisms for passing virtual functions (and perhaps non-virtual functions as well) from one system to another. We believe that these mechanisms should be independent of the hardware, the operating system, and the programming language used to create the objects. Although significant progress is being made in this area, we believe that there is still much work to be done.

6 Conclusion

Although there is still much to be learned about designing component level applications, the categorization of components and the design styles presented in this paper should prove to be a powerful guideline for those who wish to design component level applications. Even if no new components are to be created during the development of the application, the categorization of components given here will assist developers in understanding how components interact in an application.

As with any new endeavor, this is only the beginning. We believe that it will eventually be necessary to subdivide our component categories into more detailed subtypes. We also expect that new categories of components will appear in the future.

The main problems remaining are the integration of component categories into existing design methodologies, new research into component support and communication mechanisms, and research into mechanisms for supporting semi-persistent objects. The future is bright, and new developments should prove interesting.

7 References

1. "VBxtras, The Ultimate Tool Catalog for Visual Basic," <http://www.vbxtras.com>.
2. "Component Source Catalog," <http://www.componentsource.com>.
3. C. Szyperski, "Component Software: Beyond Object Oriented Programming," Addison-Wesley, Reading Mass., 1998.
4. A. Denning, "ActiveX Controls Inside and Out," Microsoft Press, Seattle, 1997.
5. "The Microsoft ActiveX Website," <http://www.microsoft.com/com/tech/ActiveX.asp>.
6. "The Microsoft COM Website," <http://www.microsoft.com/com/tech/com.asp>.
7. E. Harmon, "Delphi COM Programming," Macmillan Technical Publishing, 2000.
8. D. Doherty, R. Leinecker, "JavaBeans Unleashed," Sams Publishing, 2000.

9. B. Meyer, C. Mingins, eds., "IEEE Computer Magazine Special Issue on Components," Computer Magazine, Vol. 32 No. 7, July 1999.
10. W. Kozaczynski, G. Booch, eds. "IEEE Software Magazine Special Issue on Components," IEEE Software, Vol. 15 No. 5, Sept.-Oct. 1998.
11. P. M. Maurer, "The Virtual Design Automation Laboratory Website," <http://www.csee.usf.edu/~maurer/vdal>.
12. C. J. Date, "An Introduction to Database Systems, 7th ed." Addison-Wesley, Reading Mass., 1999.
13. OMG, "The OMG Corba Website," <http://www.corba.org>.