

1. Introduction

1.1. The Component Revolution

There have been three major revolutions in the history of computing: the stored-program computer, high-level languages, and component-level programming. The first two revolutions were textbook examples of how revolutions are “supposed” to occur. Great bodies of learned men and women contributed their best ideas. The great universities and research laboratories contributed their very best minds. Principles were elucidated, and the work proceeded along grand designs laid down by great visionaries. The people who participated in these revolutions knew that the world was watching them. They knew that they were making history.

In contrast, the component revolution happened almost by accident. Although there are many component technologies available today, the component revolution is the story of a single technology, the VBX and its heir-designate, the ActiveX control. VBX technology was neither the first nor the best (although these points are debatable), but it was the technology that convinced the world that component-level programming was a good idea.

The term “VBX,” which stands for Visual Basic eXtension, was a mechanism for adding new features to the earlier versions of the Visual Basic programming language. To create such an extension, one needed to create a run-time function library with certain standard interface functions. These libraries were stored in files that ended with “.vbx”, hence the name. Once a VBX was created, you could add it to a Visual Basic project by copying it your hard drive, and inserting a reference to it into your Visual Basic project

file. In most cases, no complicated installation procedures were needed. Once included in a project, the VBX would provide one or more *custom controls* that could be used by the Visual Basic programmer. The *control* was, and is, the mainstay of Visual Basic programming. A major portion of the VB programming involves adding controls to an application window. Standard controls could be selected from a menu of available controls, and included things like buttons, text boxes, and labels. The custom controls from the VBX could be used in exactly the same manner as the standard controls. They could be added to an application window, and code segments could be added to integrate the functions of the custom control into the program. Multiple non-interfering instances of controls could be created with virtually no effort. Except in rare cases, very different sorts of custom controls could be integrated into a single program without worrying about compatibility. Very quickly after the introduction of Visual Basic, a stunning variety of custom controls appeared providing virtually every sort of function from spread-sheets to word processing. Almost anything that was available as an independent program was also available as a VBX custom control. Instead of performing their functions in their own window, they performed them in a sub-window of a Visual Basic application.

The most important point about VBX programming is that it allowed people to program at a new level that went far beyond statement-level programming. To be sure, some statement-level programming was still necessary, but the most complex and detailed portions of a program could now be encapsulated in a collection of VBX components. The statement-level code was used primarily as “glue” to connect components together. Most of the programming practices that were used in VBX-based

programs have carried over to ActiveX-based programs, as well as to other component-based technologies.

To understand just how important this revolution is, it is necessary to realize that programming at “far above the statement level” has been a goal of programming-language theorists for many years. One conventional view was that one could achieve this goal in a series of incremental steps. The idea was that one could start with a conventional programming language, and add new, more powerful primitives to create a more powerful programming language. This new language could then be used as the basis of an even more powerful language. One could continue in this manner until the goal of programming at far above the statement level had been achieved.

Even though the incremental approach seemed reasonable, it never worked. The elusive “next step” never materialized. Many new primitives were introduced, but most of them proved to be of limited usefulness. Many new languages were created, some of which were completely different from conventional languages, but most of them disappeared in less time than it took to create them in the first place. Innovations that were heralded as the awaited “next step” turned out to be not so innovative as they first appeared, or too cumbersome for practical use. (Functional languages were one such innovation.)

One useful innovation was the shift from procedural programming to object oriented programming. Along with this change came a shift in perspective away from new primitives to code reuse. The idea was that one could program at “far above the statement level” not by using new language primitives, but by reusing major code elements created by others. In the beginning, there were some spectacular success stories that suggested

that there really was something to this idea. One of the more impressive success stories is the GUI packages that were developed for MS DOS programs. Using these packages, one could develop a sophisticated graphical user interface for a program in an environment that was quite hostile to GUI programming, and do it in virtually no time. Unfortunately there were few other success stories like this, and the disappearance of MS DOS applications has rendered even this success story irrelevant. Modern GUI operating systems do have object-oriented development packages, these packages are generally just wrappers for existing GUI functions.

That is not to suggest that object oriented programming is unimportant. Nothing could be further from the truth. Object oriented programming is an important innovation that permits one to organize programs in a logical and understandable way. It allows one to create programs that are much easier to write and to maintain than their procedural counterparts. However, object oriented programming did not change the way we write code. It changed the way we look at data and our view of the relationship between code and data. But when we write code, we still use assignments, while statements, for loops, and all the other stuff. Although object oriented programming made it easier to reuse code, code reuse is a concept that still appears more often in textbooks than in real programs.

Then along came Visual Basic with its VBX components. Suddenly everyone was reusing code, everyone was programming at “far above the statement level,” programs were being developed much faster, and they had more features than their object-oriented counterparts. What made all this happen? The answer seems to be “VBX components.”

The real question is why VBX's succeeded when other, more carefully thought out approaches did not. The answer to this question is important because it will profoundly affect the way we view components. We would like to develop a view of components that is enlightening enough to allow us to develop new component technologies without losing whatever it was that made VBX's so successful in the first place. More importantly, if we develop a thorough understanding of the VBX technology and what made it successful, we may discover that this technology is a subset of something more extensive, and ultimately much more useful.

1.2. Engineering and Computer Science

Despite the fact that we like to call ourselves computer scientists, programming is, strictly speaking, not a science but an engineering discipline. Programming does not use the scientific method, but it does use scientific and mathematical principles. Programmers create solutions to problems using the design principles that they have learned. Except in extremely unusual circumstances, they do not formulate hypotheses or conduct experiments to confirm hypotheses.

It is my belief that in our failure to discover the “next level” programming language, we have inadvertently discovered a principle that has been known for many years in other engineering disciplines. Namely, that it is difficult or impossible to design with mid-level parts. When you set out to build a fleet of cars, you don't buy your engines from Mitsubishi, your doors from Ford, your wheels from Chrysler, and your trunk-lids from General Motors. If you did, none of the parts would fit. (Auto manufacturers *do* buy parts from one another, but only for products that were carefully engineered to use these parts.) However, when building your fleet, you *could* buy some cars from Ford, some from

Chrysler, some from Mitsubishi, and some from General Motors. You would then be “engineering” your fleet of cars with finished products, not with mid-level parts.

This illustrates another well-known principle that we have rediscovered, namely that finished products *can* be used as components in larger designs. Before the VBX and other component technologies, the closest thing we had to components was the subroutine package. (I use the term *subroutine package* in its broadest possible sense.) The standard C library is an example of such a package. One midsize part that can be found in this library is the *quicksort* function. I’ve used this function many times, but I always have to look up the parameters. I’ve sometimes found it necessary to fiddle with the prototype to get it to work, and I’ve found that getting the comparator callback-routine to work properly is something of a black art. All too often I’ve said to myself, “Oh it’s a short array, I guess I’ll just write a quick insertion-sort instead.” The point being that using mid-size parts is far more demanding than using ordinary primitives, and far more demanding than using finished products.

There are more complex subroutine packages that provide features such as word-processing or image analysis. Typically an application that uses package-X must be specifically designed to be a package-X application. The program must conform to the needs of the package, because the package will not or cannot conform to the needs of the program. Needless to say, it is usually not feasible to combine two or more such packages in a single program.

In contrast to subroutine packages, components conform quite easily to the needs of a program. If you decide to add component X to your program, you can do so at virtually any point of the development without having to start over from the beginning. How you

use the component is up to you. You seldom have to worry about restructuring your program to conform to the needs of the component. (Most Visual Basic programmers would consider such an idea to be absurd!) When using components it is feasible, and quite reasonable, to combine a word processor, a spreadsheet editor, a web-browser, an animated video player, and an mp3 file player in a single application and expect the whole thing to work.

Despite the fact that a complex subroutine package might provide the same functions as a VBX or other component, the subroutine package is not a finished product. It is a partially engineered solution to some problem that must be combined with additional coding to create a finished product. Components, on the other hand, *are* finished products. Even simple components that provide nothing more than buttons or text-boxes are finished products, and despite the extensive support that must be provided by container applications, they are stand-alone programs in their own right.

It is from this perspective that we will to examine component-level programming.

1.3. The Definition of a Component

In this book we define the term *Component* to be “an independent program that can readily be used as part of another program.” This may be difficult to accept at first, because most components can run only in a host environment, and place a heavy demand for services from that environment. This seems to contradict the label “independent program.” However, we would insist that all programs place demands on their environments, and that the demands placed by a component are only a matter of degree. A C program running from the UNIX command line (for example) places demands on the UNIX shell. The shell must open the stdin, stdout, and stderr files for the program. This is

a service that goes beyond normal operating system I/O support, memory management, and other run time services, and not all programs that run in the UNIX environment are provided with this service. While it is true that the demands that a component makes on its environment are more complicated than those made by a command-line program, they are operating system services in the same sense that opening stdin, stdout, and stderr is an operating system service.

The Visual Basic run-time environment that is included with every Visual Basic program provides extended operating system services that support component instantiation and execution, scheduling and execution of glue code, and the communication interface between glue code and components. Collectively this collection of extended services constitutes the Visual Basic *Component Framework*. Every component technology has its own component framework. It is the design of the component framework that defines a particular technology.

1.4. Component Frameworks

Like any other program, a component must be loaded and executed by some external agency. It must be provided with I/O and communication services. It must be able to interact with its environment, with other components, or with both. The design of the component framework defines the basic character of a component technology.

As with components, one can view a component framework from the user's perspective or from the developer's perspective. The *user* of a framework is the Visual Basic programmer, or the web-page designer, namely the one who uses the framework to create a new application. The *developer* is the person who designs and builds the framework. These two perspectives are quite different. In the ideal situation, the user is

barely aware of the framework, and uses simple programming paradigms to set up the required component interactions. The developer, on the other hand, must be intimately familiar with the implementation details of both the framework and the components themselves. The user's simple view of the world must be translated into the complex world of the underlying implementation.

Standards for component-level programming provide specifications for both the component and the framework. For example, the ActiveX specification describes in detail, the services that must be implemented in a module for it to interact properly with its environment, and at the same time specifies those services that must be implemented by a framework to allow it to interact with ActiveX components.

There are two broad categories of component interaction, namely component-container interaction and component-component interaction. A *container* is an entity that implements the component framework and maintains a collection of one or more components. A Visual Basic program is an example of a container. Mechanisms for component-container interaction have been thoroughly standardized, but component-component interaction is generally reserved for a few specific applications. Generic standards for direct component-component interaction are rare. The most commonly encountered example of direct component-component interaction is the communication between the Visual Basic Data Control and a data-aware component.

At the lowest level, there are two ways that entities can communicate with one another. (An *entity* being either a component or a container.) There is *active communication* where *Entity A* calls a subroutine contained in *Entity B*, and *passive communication* where *Entity A* modifies a global variable that is also accessible to *Entity*

B. Virtually all component technologies use *active* communication, although it is generally possible for the user to *simulate* passive communication when necessary. In active communication, there are several recognized categories of subroutines, some of which are listed below.

Set_Property_P The user believes that P is a variable that is maintained as part of the component's internal state. The Set_Property_P function is used to assign a new value to P. P may be a real variable in the component's internal state, or it may be a virtual property whose value is computed when needed. Setting the value of a virtual property will generally trigger some complex behavior on the part of a component. This function has one parameter, the new value of the property, and no return value. Array properties will have additional parameters to specify array indices. In some technologies, return values are used to indicate the success or failure of a function, in which case the Set_Property_P would have an appropriate return value for that purpose.

Get_Property_P The dual of Set_Property_P, but used for reading the value of a property. This function has no parameters and one return value, the value of the property P. If the property is an array, the function will have one parameter for each array index.

- Event_X** An event mechanism allows a component to call a subroutine in another entity, usually the container. The event subroutine can have parameters, but generally does not have a return value. The receiving entity does not need to implement the event subroutine, in which case the it must substitute a no-op subroutine for it. Event mechanisms are typically the most complex interactions in a component technology.
- Initialize_State** This subroutine is called with a parameter that contains initialization data for the component's internal state. This mechanism is generally invisible to the user, although the user may specify the content of the initialization data.
- Save_State** This subroutine is called with a parameter that specifies a container for saving initialization data. This data will later be passed to the Initialize_State function. Neither the Initialize_State function nor the Save_State function are strictly necessary.
- Method** A method is an arbitrary function that is explicitly called by the user of the component. The function can have input parameters, return values or both. Implementing input values and return values generally requires a significant amount of effort on the part of the framework developer.

1.5. Programming Models

By far the most useful component frameworks are those that support fully programmable containers. The utility of such frameworks depends on the simplicity and effectiveness of the programming model. There are several recognized features that the programming model should support. The most basic feature is the ability to create instances of components, including multiple non-interfering instances of the same component. The model should support a “data and functions” interface similar to that of object-oriented programming. (In component terminology, data elements are called *properties*, and function elements are called *methods*.) Although a component instance is not an object in the usual sense, it is convenient for the programmer to treat it as if it were. For example, if *Text* is a property of a component named *Editor*, and there are two instances of *Editor* in the program, *Editor1*, and *Editor2*, it should be possible to transfer data from one instance to the other using the following statement, or something similar to it.

Editor1.Text := Editor2.Text;

If the *Editor* component has a method *Clear* that is used to erase its contents, then it should be possible to invoke this method on a particular instance using a statement similar to the following.

Editor1.Clear();

In other words, the Property/Method interface should be modeled after the corresponding object-oriented interface.

Finally, the framework should provide an event mechanism that permits the component to communicate with its host. Typically the host programmer would declare a subroutine to be the event handler for events of type X, coming from component instance Y. In Visual Basic, the name of the subroutine is used to associate it with the proper instance and event type. Other languages provide less haphazard methods for making such associations.

For many component technologies, the programming interface is polymorphic. That is, the properties, methods, and events that are available depend on the state of the control. Although it is possible to fine-tune the interface in complex ways, the most important distinctions are made based on whether the component is running in *design-mode* or *run-mode*, and whether or not the component is *licensed*. A component is in run-mode when the application is being executed. A component is in design-mode when it is being used to construct a new application. Unlike subroutine packages, a component can be loaded and running while an application is being constructed. This allows the component to respond to design-time changes in its properties. Certain properties may be designated as available only in design-mode, while others may be available only in run-mode. Some components may designate themselves as run-time-only, and will refuse to initialize themselves in design-mode.

Commercially available components generally require some form of electronic licensing to enable their full capabilities. Unlicensed components will generally refuse to

initialize themselves in design mode, or will run in a restricted “demo” mode. Some components also require a run-time license.

An essential feature found in most existing technologies is the ability for components to define their own interfaces. In other words, the information about what properties, methods, and events are supported by a component can be extracted from the component itself. No independent declarations are required. This permits the container development system to verify the correctness of references to instances of the component without forcing the host programmer to juggle header files.

Although the Properties/Methods/Events communication model is nearly universal in component technologies, in the future we may find an entirely different model that vastly superior to the Properties/Events/Methods model. However, as yet no one has suggested anything better. From a technical point of view, the only feature that is *required* is the ability of one entity to call subroutines defined in another.

1.6. Component Development

Although object oriented design is now considered to be an essential part of component development, earlier technologies were not object-oriented. In fact, component technology and object oriented technology are quite independent of one another. Nevertheless there have been considerable benefits gained from the marriage of the two. The most important benefit is the underlying model that is used to represent an instance of a component. Each instance of a component must have an environment that is separate and distinct from every other instance of a component. One could create a separate address space for each component, but this would not permit easy communication between a component and its container, or between two components.

Using object oriented technology, it is possible to provide each instance with its own environment by defining a *support class* to implement the environment. When a new instance of a component is created, a new instance of the support class will be created to maintain the environment. All instance data for the control can be declared as data members of the support class, and all functions that are required to implement the control can be declared as function members of the support class. In the remainder of this book, we will assume that a support class has been defined for each component.

The use of support classes can sometimes be a source of confusion for new component developers. It is easy to forget that there is a distinction between the object model of the control instance, and the object model of the support class. Although component properties are usually implemented as data elements of the support class, it is not *necessary* to do so. The support class will generally have many data members that do not represent properties of the component. By the same token, methods will be implemented as function members of the support class, but may have declarations that are different from those of the support class member. For example, in ActiveX components, the support class is generally defined in C++, while the component interface is defined in IDL (Interface Definition Language). The corresponding function declarations are different, and both are different from the Visual Basic programming model, as the following example shows.

ActiveX Method Declaration (idl):

```
[id(8)] HRESULT GetValue([in] long x,[in] long y,  
                        [out, retval] long *ReturnValue);
```

Support-Class Function Definition (C++):

```
STDMETHODIMP CSuppCtrl::GetValue(long x, long y, long *ReturnValue)
```

Visual Basic Usage:

```
X = Instance.GetValue(x,y)
```

In our examples, we will distinguish between the three points of view, interface declaration, container-language usage, and implementation.

Current standards for control development are extremely complex, probably much more so than necessary. However, we must deal with these standards as they exist today, not as we would wish them to be. Fortunately, there exist tools that permit us to build skeleton components that hide most of the messy implementation details. Without such tools, component development would be a daunting task requiring specialized training and experience. With these tools, however, component development is straightforward. We will make use of various tools to simplify development, but the reader is cautioned that these tools could change substantially in the future.

1.7. Conclusion

As with all revolutions, there are some who welcome it, and others who wish it never happened. Regardless of how one feels about it, the component revolution is an accomplished fact. Component-level programming is not “the technology of the future,” it is the technology of *today*. Component-level programming has had a profound effect on programming, particularly in the business world, and new developments in the area will continue to have a profound effect on the future of program development everywhere. It is necessary for us to make the first steps into understanding these new technologies, with the aim of both using them effectively, and improving them for the future.