Components: What If They Gave a Revolution and Nobody Came?

Peter M. Maurer
University of South Florida

Components swept through the cubicles of frontline programmers, but hardly caused a stir in the halls of academia. It's time researchers woke up and took part in the revolution.

There have been three great revolutions in computing technology during the past 50 years: the stored-program computer, high-level languages, and component-level programming. Although working programmers are well aware of this last revolution, it seems to have escaped the notice of most everyone else. Academic researchers are doing little or nothing that touches the subject, and apart from trade journals and magazines aimed at developers, publishers have all but ignored it. Yet, when we speak of component-level programming, we are not speaking of the future. The revolution has already happened, and in the academic community, nobody came.

AN EVOLUTIONARY REVOLUTION

Unlike earlier so-called revolutions—such as structured or object-oriented programming—component-level programming is a true revolution on a par with stored-program computers and high-level languages. At first glance this claim may seem outrageous. After all, the stored-program computer was a major news story that captured the imagination of the world. And development of the first high-level language launched one of the most important research fields in existence.

Component-level programming came about through an evolutionary process that allowed it to escape the notice of almost everyone except those directly involved—application programmers and people who develop tools for application programmers. They were far more concerned with meeting deadlines than with announcing their discoveries to the world.

Despite the lack of press coverage, there is no question that a revolution has taken place. Visual Basic, the first language that supported component-level programming, is now the preferred language for new development. New applications are vastly more powerful than those developed less than a decade ago, and the development of these applications takes only a tiny fraction of the time that would have been required a few years ago.

The beauty of components is their power and ease of use. Virtually anything that is available as an application program is also available as a component, and most of these components can be integrated into a new application with little conventional programming.

0018-9162/00/$10.00 © 2000 IEEE

COMPONENT VERSATILITY

The range of available components is stunning. If you are unfamiliar with component-level programming, I strongly encourage you to check out the Web sites of a few component vendors.[1-3] There are components for 3D modeling, viewing and editing spreadsheets, full-featured word processing (including spell checking), creating and editing graphics, creating and playing sound files, communicating with other computers through a modem or the Internet, geographical mapping, reading bar codes, and virtually anything else you could imagine. Because new components require little or no programming expertise, the learning curve for using new ones is virtually nonexistent.

Components have changed the way that programmers develop major applications. A traditional application consists of an integrated set of routines that solve several different problems. There are routines for handling messages and others for handling file I/O and processing data. Even though these tasks are quite different, they are often integrated in such a way that it is difficult to separate one from another. Component-level programming divides the same application into separate components, some of which are developed locally, and others that developers purchase from a component vendor.

You could argue that developers could accomplish the same thing using ordinary software packages. While this is technically true, a component tends to be more versatile and easier to use than traditional software packages, as my own experience demonstrates.

A SHORT AND SOMEWHAT PERSONAL HISTORY OF COMPONENTS

My own introduction to components illustrates why they have been so successful. Several years ago, I was developing a database system for a nonprofit organization. They needed to distribute portions of the database to several individuals who would perform data entry and other tasks in their homes. I wanted to use a popular PC-based database package, but the cost was prohibitive: The organization would have had to buy a copy of the software for each individual. It would have been possible to distribute just the runtime code for the system, but the license for the runtime-only code was also more than the organization could afford.

I explained this problem to a friend, and he said, "What you need is Visual Basic. It has a front end just like the database system you are using, and you can distribute the executable modules for free." After some research, I discovered that I could purchase Visual Basic for about one-fourth the cost of the database system's runtime license, and that I could indeed distribute the executable modules without additional cost.

I could also use Visual Basic as a front end for virtually any PC-based database system and for several client-server systems, as well as convert software from one database system to another with virtually no effort. This was like getting several thousand dollars' worth of software for free, and seemed almost too good to be true. It must have seemed that way to others too, because at about this time Visual Basic became the preferred development platform for database applications on the PC.

Controls

In its earliest form, Visual Basic was similar to several other tools that developers used to design dialog boxes. MacIntosh Resource Editor, Borland's Resource Workshop, and Microsoft's App Studio are some examples. There are many such tools that allow developers to create visually pleasing dialog boxes by drawing such devices as buttons and edit boxes directly on the dialog background.

Developers commonly refer to items that appear on a dialog box as *controls*. Buttons, edit boxes, labels, list boxes, radio buttons, and check boxes are all examples of controls. The most common controls are supported directly by the GUI, with standard subroutines to support the default behavior of the control. Controls require direct support, because dialog boxes are not important enough to warrant a lot of custom programming. More importantly, for ease of use, all controls of a certain type should behave the same way.

Although the standard set of controls is limited, it is surprisingly powerful. Virtually anything you could ever want to do with a dialog box can be done with

Further Reading

The primary programming languages that support component technology are Visual Basic, Visual C++, Java, and Delphi. Most bookstores offer a wide range of titles on how to use component technology with these programming languages.

Three Web sites offer information about specific component technologies:

- http://www.microsoft.com/com: ActiveX, COM, and DCOM are key component technologies from Microsoft.
- http://www.javasoft.com/beans: JavaBeans is proprietary technology of Sun, which hosts this site.
- http://www.omg.org: The Object Management Group (a nonprofit consortium) hosts this site, the primary source for information about CORBA.

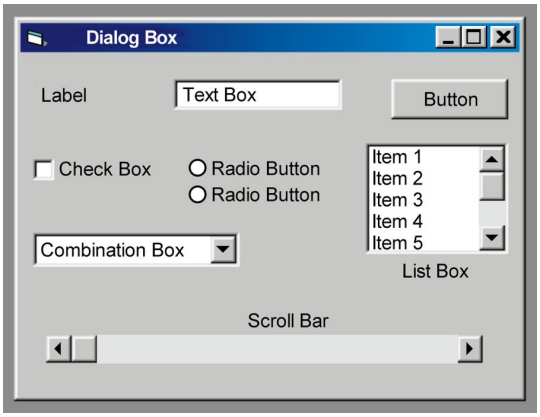

Figure 1. Standard controls for Visual Basic.

the standard controls illustrated in Figure 1. From the beginning, GUI implementers realized that the standard set of controls would not be sufficient for every application. To provide for future expansion, they included a special interface that allowed programmers to add new custom controls. A programmer would provide a set of routines to support the default behavior of the custom control, which users could then employ to support every instance of the control.

Because the standard set of controls was sufficient to meet the needs of virtually all dialog boxes, custom controls were seldom used. Radio knobs are the most common example of custom control in programming manuals. A radio knob is a substitute for a scroll bar, with the added "feature" that it is nearly impossible to use. Initially, I considered the custom-control interface to be a mechanism for creating "useless trinkets." Nevertheless, it was the Visual Basic custom-control interface that provided the foundation for the component-level programming revolution.

Visual innovations

Visual Basic introduced several important innovations, including elevating dialog boxes from mere messengers to main programs. The Visual Basic form, which is essentially nothing more than a dialog box, serves as the main routine for a Visual Basic program. (It is possible to use a more conventional main routine, but I don't know anyone who actually does.)

Visual Basic simplified the mechanism for communicating with controls; conventional programming used standard GUI calls for this purpose. Visual Basic replaced the GUI calls with a simple assignment operation. It also gave controls a set of *properties* that programmers could use to perform operations on the control. To access the text of an edit box, for example, you simply used the text property as if it were an ordinary string variable.

Visual Basic also replaced conventional GUI message processing with a collection of events that allowed the control to communicate with its environment. An example is the click event, which occurs when a user presses a button. To intercept this event, the Visual Basic programmer creates a subroutine with the name Button1_Click, where "Button 1" is the name of the button. The only deficiency was the inability of custom controls to support user-defined functions or methods, as they are more correctly called.

Visual Basic's most important innovation was an enhanced mechanism for adding custom controls by distributing them in modules with the Visual Basic extension (VBX). From the user's point of view, it was almost idiotically simple. To add a new control to your project, all you had to do was save the VBX module on your hard drive and add the filename to the project. The new controls would then appear in your list of available controls, and you could use them just like any other control.

From the control developer's point of view, VBX was a conventional program module with an unusual extension. Programmers had to include certain standard functions to enable the module to communicate with the Visual Basic environment, but in most respects, they developed the VBX module in the same way as any other module. Programming was simpler than that for an application program because the programmer could concentrate on a single task instead of juggling a complex interaction among the menus, main window, and a collection of dialog boxes.

Data control

Version 3 was the real turning point for Visual Basic. It included a new standard control known as the *data control*. Programmers could use this control to access a database table or the results of a query and hook it into virtually any database. They could synchronize the standard controls with the data control so that their contents would change when the data control moved from one record to another. The user could update the database by typing new values into the standard controls—all with little or no conventional programming.

Database access is what sold Visual Basic as a legitimate programming language. Unfortunately (or fortunately, depending on how you look at it), the set of standard controls provided with Visual Basic was insufficient to support database programming. The most glaring omission was a control that could be used to display a list of database records with automatic database synchronization. Third-party developers were quick to fill the gap, and the wide use of the Visual Basic language provided them with a ready-made market.

Almost overnight, a huge variety of custom controls appeared, duplicating virtually any function in a conventional program. There were grid controls that

could display multiple records with automatic database synchronization, text editors that duplicated the functions of major word-processing programs, graphical editors, picture editors, data-communication tools, and Internet-access components.

VBX ingredients for success

The most difficult thing to understand is why VBX succeeded so spectacularly where other, more powerful technologies did not. You can't do anything with a VBX that you can't do with a standard function library. Many of the functions a VBX provides could be provided in a more versatile way using compile-time class libraries.

VBX was so successful because of its programming interface. The only mechanism for communicating with a VBX was assigning a value to a property. Furthermore, property values had to be simple types. It didn't permit pointers, call-back functions, and complex data structures. (It was possible to have properties that were arrays of simple types, but programmers seldom used such properties.)

Although events provided a mechanism similar to call-back functions, they were too inefficient to be used for that purpose. The inherent weakness in the interface, combined with the inherent weakness of the Visual Basic language (at least in its earlier versions), forced VBX developers to use good programming practices. Furthermore, VBX was completely isolated from the Visual Basic programmer. It was impossible to "fiddle with the data structures" to make that VBX "more efficient." You either used it as it was intended, or you didn't use it at all.

COMPONENT TECHNOLOGY TODAY

The advent of modern components came with the introduction of Windows 95. Along with the new 32-bit operating system came a new type of control called an OLE control. The "official" explanation for abandoning VBX in favor of the OLE control was that VBX technology was "too married" to the 16-bit operating system to make the transition. Whether or not you accept this explanation, the OLE control introduced several innovations that changed the face of component-level programming.

The most obvious change introduced by OLE control was support for user-defined methods—however, the most intriguing changes occurred beneath the surface. OLE technology was more firmly grounded in object-oriented programming than the earlier VBX technology. Programmers could use the OLE control in a variety of environments, not just Visual Basic.

The OLE control was based on a well-thought-out standardized interface specification, as opposed to the ad hoc interface specification used for the VBX control. Unfortunately, the changes came at a heavy price: OLE controls were generally much larger and slower than their VBX counterparts. They were also more difficult to program and to install, but new development tools more than made up for those shortcomings.

The OLE control introduced several innovations that changed the face of component-level programming.

ActiveX controls

Today OLE controls are called ActiveX controls. New programming techniques have led to smaller and faster controls, and the underlying technology has led to several interesting developments. ActiveX controls are based on a technology known as the Common Object Model (COM), which in turn is based on the concept of published, immutable interfaces. Any COM module must provide the IUnknown interface, which allows module users to identify and access the interfaces supported by the module.

An interface is a collection of methods that programs use to access the services provided by the COM object. ActiveX controls must support many different interfaces, the most important of which is the IDispatch interface. This interface allows the ActiveX host access to the properties and methods of the ActiveX control.

Many applications can now host ActiveX controls. The most interesting of these is Internet Explorer. The ability to place ActiveX controls on a Web page provides users with an intriguing new way to distribute software. With this method of distribution, accessing new software is as easy as visiting a new Web site. The first time a user accesses a page, Explorer downloads and installs the required ActiveX controls on the client system. It downloads only those components that the client actually needs. Distributors upgrade software by placing a new copy of the ActiveX control on the server, and Explorer automatically upgrades users the next time they access the Web site. Of course, several new problems arise with this method of software distribution, including licensing and security issues.

Recent developments

It makes sense that the COM technology on which ActiveX controls are based has become a primary development technology in its own right. A full ActiveX control is a complicated object with many interfaces. Each control instance has a drawing surface that is either part of an existing window or implemented as a separate window. Many applications do not require the complexity provided by a full ActiveX control. There are many existing ActiveX controls that provide only background services. In such cases, programs require only a small custom interface to access the control's services.

Direct use of COM technology is an efficient way to provide a simple, efficient interface. Although there

Unlike ActiveX and VBX controls, COM and the other new technologies are designed for back-end development.

are many other ways to provide simple interfaces, COM technology's main advantage is standardization. Because programs access COM interfaces in a standardized way, any client module can query the object to discover the types of interfaces it provides. Once published, COM interfaces must remain frozen for all time, so any application that understands a particular interface can use the object. Even if the module is redesigned, any module that uses the interface must be able to use the redesigned module without change.

Dynamic instantiation. Another advantage of COM objects is that they can be dynamically instantiated. This is difficult to do with ActiveX controls because an ActiveX control must be contained in a window. When the main window containing the ActiveX control is instantiated, the control is instantiated at the same time. In contrast, COM objects are not generally associated with a window, so dynamic instantiation tends to be the rule rather than the exception.

Dynamic instantiation is useful in developing software for electrical-design automation. For instance, we have a collection of circuit simulators that provide a range of trade-offs in accuracy and speed. Using COM objects makes it easy to maintain a list of installed simulators and allow users to choose the particular simulator that meets their needs. Only those simulators that are actually used need to be instantiated. It is possible for users to invoke simulators that were not even in existence when the original software was created. This would be difficult to accomplish with ordinary ActiveX controls.

Universal standard. Unlike other component technologies, COM is not limited to the PC. Designers have implemented it on virtually every major operating system, making it a truly universal standard. Furthermore, developers can build COM components in almost any language, including Java, C, Pascal, and even Cobol. In addition to platform independence, developers have created new standards that allow components to communicate with one another even though they reside in different processes or on different systems. This new standard is known as distributed COM (DCOM).

DCOM is just one of several distributed-object technologies. Other prominent members of this family are the Common Object Request Broker Architecture (CORBA) and Java Distributed Objects. It is not yet clear which, if any, will become dominant.

With any of these distributed object technologies, programmers can create objects on one machine and access them from a different machine or from a different address space on the same machine. In some cases, programs pass objects by reference, which requires the original process to host all operations on the object. In other cases, objects are physically transferred from one machine to another through a process called *marshaling*.

Marshaling software converts objects to a machine-independent form and transmits them between address spaces or between machines. The most sophisticated marshaling software is capable of handling complex objects containing pointers and linked lists.

The differences between COM and DCOM are minor, so developers can easily convert applications built around COM to distributed applications. Although there is enormous potential in distributed-object technologies, it is not yet clear what impact they will have, or how they will affect software development.

THE FUTURE

Unlike ActiveX and VBX controls, COM and the other new technologies are designed for back-end development rather than user-interface design. Although using components for back-end development is not new (ActiveX and VBX controls have been used for this purpose through a feature called invisible at runtime), it is not yet clear that component-level development will have the same impact on back-end software as it had on user-interface design. COM and other distributed-object technologies are more efficient for back-end design than either ActiveX or VBX objects, which are somewhat more difficult to use. Because objects must be explicitly instantiated and initialized, it is not clear that back-end components have any inherent advantage over more traditional software packages.

To be successful at the back end, component-level programming must be closely integrated with the concept of distributed objects. My experience using ActiveX and VBX objects as back-end components indicates that a traditional application should be viewed as a series of objects, with components to perform transformations on objects. This view leads to a natural decomposition of applications into their component parts.

For example, you would view a compiler as an application based on a series of three objects: a source file, a machine-readable program description, and an object module. In this view, the compiler would contain two independent components: a parser and a code generator. The parser's function would be to transform source files into machine-readable descriptions. Because components must pass the machine-readable descriptions to one another, these descriptions must be carefully standardized. Although the parser may recognize C++, VHDL, or some other well-known language, the code generator does not need to know anything about the language. Its scope is confined to the standard objects created by the parser.

You could replace the C++ compiler with a Cobol compiler without affecting the rest of the compilation process. By the same token, the parser output does not need to be fed into the code generator. Instead, it could be used for an interpreter or for a macro processor. In this way, you can reuse components based on standardized objects for purposes that may not have been envisioned when they were first created.

Semipersistence

Back-end component reuse depends on how well the component supports standardized distributed objects. This point is so important that my research team has come to view development of a new application to primarily be an exercise in developing new standard objects. Developing each component is an independent project. Because standardized objects can have a lifetime much longer than that of the software creating them, they possess some measure of persistence even if they are never written to nonvolatile media. I call this property *semipersistence*.

To be semipersistent, an object must

- be standardized,
- have a lifetime that is independent of its creator, and
- normally reside in volatile storage.

Because semipersistent objects can be transferred from one computer to another, they could—at least in theory—exist for years without ever being written to nonvolatile storage. To illustrate the power of semipersistent objects, consider the scenario illustrated in Figure 2, which is based on work currently in progress at the University of South Florida.

The project began with specification of the semipersistent CNetList object. This object can represent circuit design details at either the logic or transistor levels. All components in our system are either CNetList producers or CNetList consumers. The components are unaware of one another's existence. The first CNetList producer was a parser for the FHDL (Florida Hardware Design Language) language. This parser currently serves as the primary front end for our system. My research team has created several CNetList consumers, most of which are simulators. We also completed work on a CNetList-to-VHDL translator, and are now creating several physical-design automation tools.

Although our system uses an FHDL front end, it would be incorrect to characterize it as FHDL based. The system is, in fact, based on the CNetList object. Any tool that creates CNetList objects can be used as a front end. For example, our schematic capture tool can serve as an alternative front end for the system, and we are planning several other front-end tools.

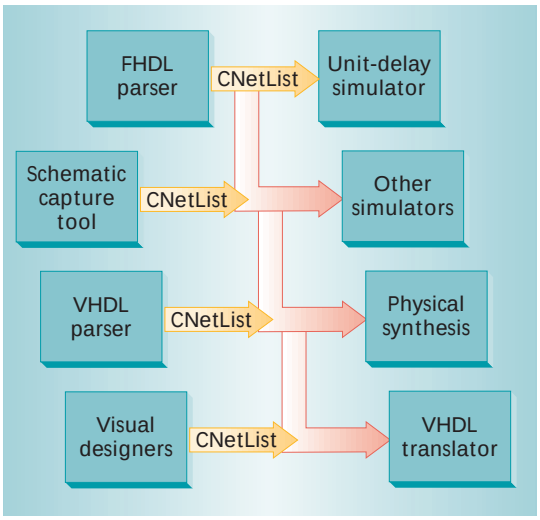

Figure 2. The CNetList object is a standard format for presenting circuit data. It exhibits semipersistence because it can be infinitely reused by several different components. These components include various parsers and design tools.

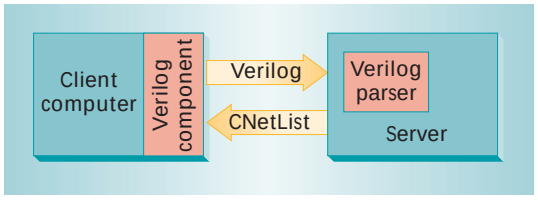

Figure 3. Remote object access made seamless by a component. A company could provide a service component to a customer, which would reside on the customer's client computer. This component communicates with the Verilog parser, which resides on the company's server. To the programmer, it looks as though the Verilog translation occurs on the client.

Marketing flexibility

The use of semipersistent objects is more than a versatile design style for locally developed software. Let us first imagine that the CNetList has become a widely accepted industry standard for representing circuits. Next suppose that a company decides to create a Verilog parser—which can serve as a front end for several different systems—but is not interested in providing anything else. Rather than marketing this new front end in a conventional way, the company decides to provide a Verilog-to-CNetList service, as shown in Figure 3.

To the client, the Verilog service appears to be just another CNetList front end. The Verilog company provides this front end in a stub component that runs on the client system. This component provides necessary communication services and seamlessly integrates the service into existing software. To the programmer, it appears that the Verilog-to-CNetList translation occurs on the client system. The company saves expensive distribution costs, and gains some protection against software piracy. It can also bill clients on a per-use basis, making the service more attractive to infrequent users and providing frequent users with more flexible billing options.

Solving transmitting problems

Programmers also can employ semipersistent objects to solve an inherent problem with transmit-

ting objects from one system to another. Although sophisticated marshaling programs can handle pointers, lists, and other sorts of complex objects, only the data portion of an object passes from one system to another. Any methods required by the object's class must already reside on the client system. This turns out to be a severe restriction for polymorphic types.

Suppose a client requests an object of class A from server S, and that class A has several virtual functions. The rules of object-oriented programming would permit server S to return any object derived from S, including one whose virtual methods have been redefined. When the virtual methods of the object S parent class are called, the methods redefined by the derived object must be called. The code that calls the virtual method does not need to "know" about the derived class.

Using semipersistent objects, a service could pass a partially compiled form of the new methods to the client system, along with the object itself. The client could either complete compilation, or execute the methods interpretatively. The programmer could write redefined methods in any available language, including one that is not available on the client system.

I believe that back-end components will eventually prove to be just as powerful and useful as today's user interface components. However, back-end components usually deal with complex data structures, making it more difficult to standardize communications between components.

The success of back-end components depends on specifying and accepting standardized objects for various tasks. It must be possible to pass complex objects between different computing systems, even when these objects have been extended with new virtual functions. Future research should concentrate on splitting standard applications into components, specifying standardized objects for various applications, and creating system-independent representations for polymorphic types. Although object-oriented programming seems to mate well with component-level programming, it is likely that the marriage of the two will change our view of both.

Component-level programming still offers enormous potential for growth, and once the area stabilizes, a wave of new ideas will arise from the research community. The prime research opportunities are in extensions of object-oriented programming to better support the component-level view of objects, the division of standard applications into components, and the standardization component communication. Methods for sharing components and objects among incompatible operating systems and hardware are also areas ripe for innovation.

Not only has the component-level programming revolution already happened, it promises to be the wave of the future. From where I sit, the future looks bright. ✷

References

1. VBxtras... The Ultimate Tools Catalog for Visual Basic; http://www.vbxtras.com.
2. ComponentSource: The Definitive Source of Software Components; http://www.componentsource.com.
3. Programmer's Paradise; http://www.pparadise.com/Paradise/home.asp.

Peter M. Maurer is an associate professor in the Department of Computer Science and Engineering at the University of South Florida. His research interests include VLSI design automation, software design, and object-oriented programming. Maurer has a PhD from Iowa State University. Contact him at maurer@csee.usf.edu.