

20. The Future

20.1. Prerequisites and Objectives

Before starting this chapter you should have:

1. A thorough knowledge of the material of Chapters 1-19.

After completing this chapter you will have:

1. Some knowledge of future trends in component level technology and design.

20.2. Introduction

Although Component-Level programming is enormously successful, I believe that it is still in its infancy. As our experience with Component-Level programming grows, I believe that we will see significant changes in existing technologies, and perhaps entirely new technologies coming into existence. In this chapter I will discuss some things that I believe should have the highest priority.

20.3. Direct Compiler Support

Few programming languages provide direct support for Component-Level programming. Although we have used Visual C++ extensively in our examples, the only thing “visual” about Visual C++ is its name. In most respects it is an ordinary C++ development system. In C++, regardless of the compiler, it is not possible to declare a variable *X* of type “TextBox” and then access the Text property *X* using syntax of the form “*S=X.Text;*”. Such features need to be incorporated into the C++ standard.

Java has a similar problem. Elaborate mechanisms must be used to access properties, fire events, and to process events. Both C++ and Java would benefit by borrowing a few features from Visual Basic.

At present, Delphi is the language that best incorporates Component-Level programming into its structure. The language not only supports direct access to components but also provides language features for the development of components. In Delphi it is possible to declare a class variable to be a property, and it is even possible to declare virtual properties that have access functions but no variable. Event declarations could use a bit of work, but in general the features of the Delphi language greatly simplify the development of components. Both C++ and Java could benefit from the incorporation of such features.

20.4. Remote Objects

Because Component-Level programming is inextricably tied to Object Oriented Programming, any research that improves our ability to handle objects remote or otherwise will directly benefit Component-Level programming. Remote access to objects is one of the most exciting developments in the area of Object Oriented Programming. Although the data portion of an object can be passed between different address spaces and even between different machines, objects are not just data. Objects are data plus behavior. Unfortunately there is no simple answer to the problem of transferring functions between different machines, especially machines with incompatible instruction sets. I believe that a combination of just-in-time compilation of object methods and independent caching of both object data and functions will contribute to the solution of this problem.

20.5. Environmental Services

In existing component systems, the environment provides a number of services to the component. At the very least, the environment must create component instances. If the component requires a window, it must provide the window and the necessary support functions. In addition, the environment must implement the mechanisms used to properties, methods and events. Events place the most strenuous demands on the environment. When a component declares an event, this is a demand to the environment that a particular function be created. Furthermore, this function must be different for each instance of the component. In terms of objects, event declarations are a demand by the component that the environment perform the following steps.

1. Create a new class for the component. This class will contain a number of functions, but no variables. The functions correspond to the events defined for the component.
2. Create an empty body for each function.
3. For each instance of the component, derive a new class from the component class and override a number of functions to provide unique behavior for the instance.
4. Create one instance of the derived class to handle the events for the component.

It is our contention that since a component can demand that one sort of object , there is no reason why other sorts demands should not be made as well. In a sense, we can

consider the method for handling properties and methods to be a demand on the environment. Each component has an associated class containing data members that correspond to properties and function members that correspond to methods. For each instance of a component, the environment must construct an object of the corresponding type.

In the future, we expect that components will be able to make more complex demands on their environment. For instance, let us suppose that components are permitted to make three different types of demands, a demand for a function, a demand for a variable and a demand for an object.

When a variable is created in response to a component's demand, it must be created within the proper scope. Several different scopes should be available. For example, if the scope is "Instance" then every time a new instance of the component is created, a new instance of the variable should also be created. If the scope is "Component" all instances of the component share the same variable, but if different components use different variables, even if the variables have the same name. If the scope is "Global" then all components that demand the variable will share the same variable. There should also be mechanisms for creating hierarchies of scopes within the environment.

If the scope is "Group" then the variable will be shared by all components in the same group. (A component group is a mechanism that will be used for defining a hierarchy of scopes within the environment.)

It should also be possible to specify the location of demanded variables. The location can be "Internal," which will make the variable equivalent to an ordinary property, or "External" which requires that the variable be explicitly created and maintained by the

environment. If the location is declared to be “Remote,” The variable will be maintained within a different component. (We believe that this is a more systematic way to handle component/component interactions such as those performed by the Visual Basic Data component.)

Demands will probably also specify the variable’s implementation, “Nullable” or “Required.” If a variable’s implementation is “Nullable” the container is free not to create the variable. All write accesses to the variable will be discarded, and all read accesses to the variable will return a specified default value.

Like variables, function demands will have a specified scope, location, and implementation. A function demand that has scope “Instance,” location “External,” and implementation “Nullable” corresponds to the notion of an event. A function demand that has scope “Instance,” location “Internal,” and implementation “Required” corresponds to the notion of a method. (When a function is declared to be “Nullable” this implies that the environment is free to create an empty body for the function.)

These concepts will probably be extended to include object-demands, which are a demand for an object with specified variables and functions. It is possible that each member of the object will have its own set of specifications, so an object could be created in “mix and match” fashion.

20.6. Layered Standards and Protocol Independence

It has been recognized for many years that communication protocols should be specified as a set of hierarchical layers. It has been further recognized that the layers of a standard are independent of one another, and the implementation of a particular standard

at one layer does not preclude the implementation of a different standard at a different layer.

There are actually three distinct layers that appear in component-level programming, although in all existing technologies, these layers are blended together in such a way as to make them very difficult to pinpoint. At the lowest level is the interprogram communication layer that is used to implement properties methods and events. In the ActiveX technology, this consists of the COM technology. The next layer is the host-component interface, which consists of properties, methods, and events. These two layers are actually distinct and should be independent of one another. It should be possible to implement an ActiveX look-alike based on message passing, shared memory, or some other interprogram communication mechanism. The third layer is the program layer. At this level a program consists of a set of Accessors, Editors, Service Wrappers, UI Widgets, and so forth. Much of what we have studied in this book is an attempt to bring some organization to the third layer of the component technology. As we have pointed out several times, the third layer is independent of the other two layers.

20.7. Conclusion

Component-Level programming is the “stealth revolution.” It developed, almost overnight, as a new and powerful way of organizing complex programs. It has become an essential tool for the development of new programs of virtually every kind, and its use grows every day. The lessons of Component-Level programming go beyond any one technology or programming language. If ActiveX and C++ were to vanish from the Earth, new languages and technologies would instantly step in to fill the gap. Despite the

success of Component-Level programming, there are many new developments waiting on the horizon. Regardless of what the future brings, it is certain to be exciting.