On To Components

Bertrand Meyer, EiffelSoft

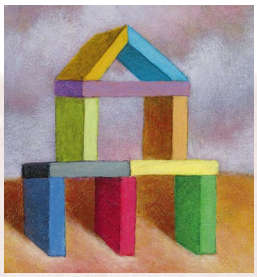
With this issue, the Object Technology department becomes Component and Object Technology. Although component technology has figured prominently in earlier columns, we felt we had to go further. Excitement about components runs high in the computing community. In a recent survey of the *Computer* readership, components placed at the very top of the topics of interest. Note, by the way, that *Computer* will devote its July issue's theme to component-based development.

All the buzz about components results from the general realization that

- software development is in trouble, and
- the kind of breakthrough we need can only be achieved through the economies of scale that result from reusing other people's creations.

It is not utterly negative to emphasize the first point. Software developers have contributed much to society. By and large, most software works much of the time. But there is a general feeling that we can't continue using our current techniques. The year 2000 mess is only one result of poor software engineering. The industry has largely ignored the lessons of Y2K ("The Opportunity of a Millennium," Nov. 1997, pp. 137-138). In cases like these, components appear to be part of the solution.

Editor: Bertrand Meyer, EiffelSoft, ISE Bldg., 2nd Fl., 270 Storke Rd., Goleta, CA 93117; voice (805) 685-6869; ot-column@eiffel.com



Excitement about components is gratifying to those of us who have been advocating component-based development for years.

OBJECT TECHNOLOGY, COMPONENT TECHNOLOGY

The change of this department's name indicates a broadening of its scope, not a reversal of its course. The phrase "objects are dead, long live components," although a good attention-getter, does not make much sense technically. All the evidence suggests that successful component technology must build on object orientation. For one thing, no one knows how to build complex, mission-critical software today without the help of OO techniques. But even more importantly, most of the fundamental ideas that define object technology are just as fundamental to any successful component development:

- Information hiding and data abstraction—to separate component implementation from component interfaces.
- Polymorphism and dynamic binding—to allow for dynamic adaptation of components to actual client needs.
- Design by Contract—to make sure that components are properly specified and validated.
- Inheritance—to organize components in rational hierarchies.

More generally, I have found that many of the techniques developed in connection with reusable class libraries and frameworks are of great interest to developers of coarser grained components such as COM and CORBA. Examples from my book *Reusable Software* (Prentice Hall, 1994) include the open-closed principle; the option-operand separation principle; the command-query separation principle; systematic component naming conventions; and systematic use of carefully crafted assertions (in particular, invariants).

It's instructive to see these techniques being rediscovered independently in the context of component development.

WHAT IS A COMPONENT?

There is no generally accepted definition of components. Here is a possible one. A software component is a program element with the following properties:

- The element may be used by other program elements (*clients*).
- The clients and their authors do not need to be known to the element's authors.

The first property excludes programs meant only for use by humans or, as with embedded software, by nonsoftware systems. The second property excludes the simple case of a module that is used by other modules—a subroutine in a traditional program, an Ada package, a class in an OO system—but without the fundamental requirement of general reusability. A true component must be usable by software developers who build new systems not foreseen by the component's author and who are not personally known to that author.

With respect to the first property of my definition, there is no requirement for the software to be usable *only* by other program elements. It is perfectly possible for

a program to be usable both by humans and by client software. An example is Microsoft Word, which is certainly meant to be used by humans, but is also available as a COM component for use by other software.

The definition immediately evokes an analogy with engineering components, in particular, electronics components. Like all analogies, it can lead to confusion if we forget that it is only an analogy. But we can indeed learn a lot from the experience accumulated by our hardware colleagues with electronic components.

VARIETIES OF COMPONENTS

There are many ways to classify components. Here is a review of components from four orthogonal viewpoints: level of software process task, level of abstraction, level of execution, and level of accessibility.

Level of software process task

The first classification addresses the software process activity to which the component applies. We may have an analysis component, which takes advantage of reusability for system modeling, a design component, or an implementation component that is an actual executable piece of software ready to be integrated in a working software system. Design components are also known as patterns.

I am using the term "software process task" rather than "process step" because in the seamless, reversible process promoted by object technology, tasks such as analysis and design are not really separate steps but activities that may intervene at different times.

Level of abstraction

The level of abstraction describes the nature of components in terms of the abstraction level they encapsulate. A component may cover:

- A functional abstraction, as with subroutines and functions in traditional libraries, each of which covers one particular function.
- A casual grouping, as with Ada packages or C files, which, unless used with precise methodological guidelines (such as data abstraction), can be used to gather arbitrarily related elements.
- A data abstraction, as with classes in OO languages, each of which covers a type of object.
- A cluster abstraction (or *framework*), which covers a set of related data abstractions intended to work together according to preset schemes: examples include EiffelBase, the C++ Standard Template Library, and some Smalltalk libraries.
- A system abstraction, which is the case of coarse-grained binary components such as some COM and CORBA components. Microsoft Word, used as a component, falls into this category.

Level of execution

The level of execution derives from the time when the component is integrated into a software system. Components may be *static* (integrated at compile time or link time and not changeable without a recompile); *replaceable* (like static components, but with variants substitutable dynamically); or *dynamic* (integrated at execution time).

Level of accessibility

The level of accessibility characterizes components by the form in which they are available to client authors:

- Interface description only, no source available. Many commercial components are distributed in this form.
- Source only, little or no information hiding. To use the component, at least for advanced uses, you must read its source text; this is sometimes the case with some of the free software available on the Internet.
- Information hiding, with reuse through the interface and source available for inspection, discussion, and correction.

THE TROUBLE WITH COMPONENTS

The industry is fascinated with components. You must go back ten years, to the time when most people first came across OO ideas, to find a comparable level of excitement. This excitement is largely justified and is gratifying to those of us who have been advocating component-based development for years.

But all the buzz shouldn't make us forget that even if components—I would say object-oriented components—are part of the answer, they are not the entire answer. Instead, they raise new questions, the foremost of which can be focused on a single topic: quality. Once the fascination with components subsides, every CIO and project leader will realize that the quality of a component-based application is defined by the quality of its worst component. This realization may be painful. If a project is a mess, it's the project leader or the CIO who will be blamed; "It's the fault of those COM controls we use" will not be a useful excuse.

So the two-step reasoning cited at the beginning of this column omitted a crucial third step: Components are worthless, and quite possibly harmful, without an impeccable guarantee of quality.

Only recently has the problem of component quality come to the fore, with the appearance of such articles as Elaine Weyuker's discussion of testing components (*IEEE Software*, Sept./Oct. 1998, pp. 54-59). And this department has, of course, repeatedly engaged the issue with articles on the Ariane disaster (Jan. 1997), the Trusted Components project (May 1998), and many others emphasizing Design by Contract and component-specification techniques.

The success of component-based technology in hardware has only been possible thanks to precise techniques for specifying components and, most importantly, to exacting techniques for quality assurance and quality control. But the realization that we are far behind in our efforts to extend this success to software components has yet to reach the software community at large. When it does it will be sobering.

It would be a pity if the move to components were to falter because of our insufficient attention to quality. We shouldn't let that happen. ❖