BERTRAND MEYER

What to Compose

Going beyond the definition of components as units of composition requires asking what and how we can compose.

I hope you are not expecting a shouting match between Clemens Szyperski and myself regarding his rejoinder ("Point, Counterpoint," Feb. 2000, p. 62) to some of my views on components ("The Significance of Components," Nov. 1999, p. 57). I appreciate both Szyperski's points and the *Software Development* editors' insights in setting up this multi-voice column. It's not a fight, but an opportunity for each reader to gain (I hope) a better understanding of the issues and to arrive at his or her own conclusions.

A small correction: I didn't write that "binary components are about information hiding." Components, binary or not, are about more than that. What I did write is that information hiding is key to the attraction of binary components, in particular for people who have been working in languages that do not properly support information hiding at the source level.

Binary: How and Why?

Let's probe further what components are really about. Szyperski notes that components foster not just reusability but also extendibility and "evolvability." Quite true, but improvements in reusability improve these other two properties anyway. He reiterates his view that only binary components qualify as components. Binary components have indeed taught us the importance of a number of points not fully grasped by the original object-oriented movement; most important, as Szyperski is right to insist, is composability—dynamic as well as static. How much does it matter that these components be source or binary? Before we can answer this question, we have to be sure that we understand what "source" and "binary" mean. Here I must confess that I don't quite know any more. In the good old days (a long, long time ago—1992, perhaps?) "source" meant something like C or Pascal, and "binary" meant code for some processor. But now we have machine-independent bytecode and scripting code, which are supposed to count as binary. Then there are high-level languages processed by Just-in-Time compilers, or, like ISE Eiffel, compiled into C and, optionally, bytecode or a mix of the two; where do components written in these languages fit?

It's not a question of platform portability: Visual Basic or COM components are platform-specific.

It can't possibly be a question of speed of access. As I am typing this article, I have cnn.com on my other workstation, and the LiveCam that I tried to bring up five minutes ago still displays "This feature requires Java. It will take a few minutes to load. Thanks for your patience." (Thanks for whose patience? Moi? I don't have any such thing. Patience is for others, such as my editor when I am past deadline for my next column.) During that time I could have compiled a sizable Eiffel application, so how does binary help me?

It's not a question of interpretation vs. compilation: A slow interpreter has no advantage over a fast, platform-aware compiler. Even a slow compiler might do, in fact, as long as it is incremental; a compiler starting from source text could work in a kind of RealAudio way, taking its time to start and then providing the illusion of regularity through buffering. If the generated code is fast, we may gain overall.

It is not about being self-contained. Many components have dependencies on others. Some of these dependencies are static; it would be naïve to think that components are any less immune than other approaches to "needed element not found" errors. This morning I had the amusing experience of hitting the same problem twice within a few minutes under completely different guises; first because a program that I was trying to link was missing some externals; then, when my browser couldn't display a page because it was missing a plug-in. Component-based approaches and source-level composition face the same issues here. Other dependencies are dynamic, with both the same benefits and the same problems with which DLLs have made us familiar.

Deployment

Is a component just a "unit of deployment"? Perhaps, but this definition is suspiciously broad. After all, any program in the traditional sense is a unit of deployment. Object technology has brought to light—and tried to free us from—the limitations of the traditional view of a program as an executable that does one thing. Beyond this simple view, object technology introduces the notion of class, providing a number of well-specified operations (commands and queries) on a certain data abstraction. We can take a program and make it into a component, but unless it is a trivial one-input/one-output program, we will need to "componentize" it: Wrap it into a hull, with a set of openings providing to the rest of the world the set of operations that we expect the program to perform for us on request. Any idea of what form such hulls would have? It doesn't take long to realize that they will be very much like classes. Even inheritance can naturally come into the picture. Oh, and by the way, we have just reinvented COM and CORBA.

This was only the case of a component derived from a legacy program. If we move forward and design new components, there seems to be little doubt that classes will provide a convenient

Continued on page 71

Bertrand Mayer is the Santa Barbara, Calif.-based author of Object-Oriented Software Construction, second edition *(1997 Jolt Award). His home page is at http://eiffel.com.*

Continued from page 59

and effective means of encapsulation.

From Classes to Components

So we are not taking much risk in asserting that classes provide the right form of components. This does not mean that components and classes are the same thing, if only because not all classes are suitable as components. It's obvious that if you pick a class at random from an object-oriented system, it will usually not (as Szyperski points out) yield a good component. Legacy programs are not the only ones that need to be componentized; it's true of object-oriented systems as well. The big difference, though, is that the process is much easier, since the necessary mechanisms of data abstraction are already in place. In favorable cases, you already have one or more classes that have been designed as interfaces into the system for the rest of the world; these will form the basis for the componentized version. If such a class doesn't exist, you will have to write it as a bridge pattern establishing a link with the facilities—implemented by other classes—that you have chosen to make part of the component. For the system designer, this process is fairly straightforward.

In the case of ISE's EiffelCOM library (http://www.eiffel.com/products/com), which among other things includes a Wizard to generate COM components from Eiffel, users were initially expected to write interface classes in COM's Interface Definition Language (IDL). But experience has shown that it is better to provide (in the latest release) an Eiffel-to-IDL translator. In many practical cases, you still have to write a bridge class, but you write it in your programming language of choice, providing—among other benefits—easy access to the other classes of the system. Then you let the Wizard translate it into the appropriate IDL. This approach seems generalizable: Rather than use the usual IDL to programming language compiler, let people use their familiar tool to produce interface classes that then serve as the basis for componentization of the surrounding system.

Client-Oriented Software

All this doesn't define "component," but it helps us get closer to a good definition. Components are (in the words of my colleague Christine Mingins from Monash University) "client-oriented software." The two basic conditions for a software element to be considered a component are that it be:

- Usable by other software elements. This excludes a program in the traditional sense that is meant to be used by humans or non-software triggers—unless it has been componentized, meaning precisely adapted for use by other software.
- Usable by software elements whose authors are unknown to the component's authors. This excludes the case of routines, classes and other software elements used by other parts of the same software. A component must be of interest to a broad range of "clients" not directly connected to the original authors.

These requirements, modest as they may seem, immediately lead to several others (see the "Seven Criteria for Components" sidebar):

- A component must include a specification of all its dependencies: hardware and software platform, versions and other components. Otherwise new clients won't be able to make good use of the component without going back to the original author.
- For the same reason, a component must provide a precise specification of the functionalities that it offers.
- The component must be usable on the sole basis of that specification, without access to non-interface information (such as the source code even if it is available). This leads in particular to the information hiding requirements discussed in my last column.
- Components must be composable with other components, since a single component is not very exciting and certainly does not justify talking about component-based development. In practice, this means that a good component will usually be part of a more general component framework with a clear overall architecture, style and standard design patterns.
- The process of integrating a component into the systems that use it should be fast and smooth.

SEVEN CRITERIA FOR COMPONENTS

1. May be used by other software elements (clients).
2. May be used by clients without the intervention of the component's developers.
3. Includes a specification of all dependencies (hardware and software platform, versions, other components).
4. Includes a precise specification of the functionalities it offers.
5. Is usable on the sole basis of that specification.
6. Is composable with other components.
7. Can be integrated into a system quickly and smoothly.

Varieties of Components

The last point—along with information hiding, as pointed out in the earlier column—is one of the arguments for binary components. But it does not imply binary components. For example an on-the-fly compilation mechanism can achieve results which turn out, as seen from the outside, to be fast and smooth enough. These are relative criteria, not absolute ones; if my compiler generates code faster than it takes to start the Java Virtual Machine, why should I care that I got my component in source code, Eiffel-generated C code, bytecode or machine code? Dependencies are also not a separating factor: there is no fundamental difference between a source component's dependence on a compiler or other translator, a machine-specific binary component's dependence on a certain hardware architecture, and an interpretable component depending on a certain virtual machine.

Binary vs. source is by far not the only dimension of choice. How do we access components? How do we pay for them, if at all? Are they platform-specific (even in a broad sense of the term platform) or platform-neutral? Is there built-in versioning? All these and others are relevant classification criteria.

One issue that I won't address here

Continued on page 74

Continued from page 71

is state—which Szyperski describes as incompatible with the notion of component—not because I necessarily disagree but because I don't quite understand his point yet, since under the heading of "state," he actually discusses staying away from global variables. I certainly concur that global variables aren't desirable in Component-Based Development or elsewhere, especially in object technology. But then again, I work in a language that doesn't support global variables—for all kinds of good methodological reasons—and the question doesn't even arise. It might arise if we interpret "global variable" in a broader, non-language sense, but then we need Szyperski to explain further what he means by global variable. Are Windows Registry entries, for example, global variables, and if so should components refrain from using them? If not, what is a global variable, or at least the kind of global variable we should stay away from in components? I am sure we will get answers to these questions, but there are others that are more pressing.

Composition Requirements

Reusability, extendibility, "evolvability"—all this is great. But wait a minute: what about reliability? How do we know that the components we try to compose (remember, components are about composition) are composable? What indeed do we want to compose?

Let me tell you the story of my blue phone and my red phone. I have two almost externally identical cellular phones—you know, the cute Nokia phones that allow you to choose (well, buy) the cover color you like. I use my blue cell phone to make calls in the U.S. and Canada and my red GSM phone everywhere else, and I have a charger for each. So there I was last month in beautiful Melbourne, Australia, conscientiously plugging the charger for the red phone into the electrical outlet, except that was the charger for the blue phone. (Strangely enough, they don't have color covers for the chargers.) U.S. current is 110V, and elsewhere it's 220V; needless to say, I need a new charger.

As a general rule, in electronics and elsewhere, pluggable components are only pluggable to the extent that they satisfy the specifications of what we plug them into. If they don't, you can't expect much. It will not always do to your component what it did to my charger, but it won't usually work. Try plugging an audio cable into your PS/2 port or a Sun Type 5 keyboard into a PC keyboard slot.

Reusability, extendibility, "evolvability"—all this is great. But wait a minute: What about reliability?

Software is no different, except in software we don't have specifications for the plugs and the outlets. Actually that's too strong. In most modern component frameworks—such as the Interface Definition Languages of COM and CORBA—we can rely on some type specification for the arguments, but it's not enough. We badly need semantic specifications as well. Type-only specifications are like diameter specifications for the electrical plug: The plug may fit even if the voltage is wrong. (Now I have to admit it. Yes, I had to interpose an electrical adapter—alas, not a converter, just the physical adapter—to plug the U.S. charger into the Australian outlet. Don't ask. I must have been jet lagged. Besides, it works for my laptop and my portable printer, so I put in the adapter almost without thinking. But this reinforces my point: type conformance is not the answer without semantic conformance, too.)

Even though we are in the software business, not the electrical-plug business, the issue is the same. Assume you have a date component written a few years ago and it gives you a mechanism to set the date, with a "year" argument that expects an integer. What may you pass to it: a number between 0 and 99, and, if so, does 24 mean 1924 or 2024? A number between 0 and 9999? Either of the above? What if the number is not within bounds?

Or take a component that, given a string representing a site's URL and another representing a search phrase, will search the site for the phrase. The type specification is simple enough, but to use the component safely and effectively you need the semantics: what happens if you can't reach the URL? What if the URL is ill-formed—but then what is a well-formed URL? Will a missing "http:" be automatically filled in? A missing "www"? Does the search exclude files listed in the "robots.txt" file as specified in the Robot Exclusion Standard? Does it only look at .htm and .html pages, or does it peer into Active Server Pages, Javascripts, style sheets and PDF documents? Will an attempt be made to search password-protected pages (one hopes not)? What should the search phrase look like, and is there some kind of query language ("and," "or" and so on)? In what format will answers be returned? Answers to these questions are a critical part of the component's specification; we may not be able to express them all formally, but the current situation where we can essentially express none is intolerable.

Contracts for Components

We can't seriously have components without contracts. To start answering the question: "What can we compose?" we can say: "At the very least, contract-equipped components."

Szyperski, in fact, explains this very well in his book *Component Software: Beyond Object-Oriented Programming* (Addison-Wesley, 1998); and one may also note that he is well known in language circles for his work on Sather, an Eiffel-like language with built-in Design by Contract mechanisms, devised in the mid-1980s at the International Computer Science Institute at Berkeley. To me, this makes some of his column's comments surprising. For example: "Classes rarely 'pay the price' for being fully explicit about what they offer and what they require." Rarely maybe, but not for programmers using languages with Design by Contract! True, *Component Software*'s contract chapter relies for its contracts on ad hoc comments in a noncontract-aware language and doesn't mention

languages (such as Eiffel or Sather) where the contracts are part of the basic fabric of programming, tied in with good design, automatic and precise documentation, goal-directed debugging, and exception handling. That's hardly representative of the actual use of the concepts.

Not to claim, by the way, that the contract mechanisms introduced in an Eiffel spirit in my book *Object-Oriented Software Construction* (Prentice Hall, 1997), necessarily transpose identically to component-based development. Its chapter on concurrency and distribution already analyzes some of what needs to be adapted, but there is more work to be done on Design by Contract in the context of independent, possibly distributed components.

The solutions must be realistic; they must be language-independent, at least for a broad class of languages—such as C, C++, Java, Visual Basic and of course, Eiffel (where the framework is already present); they must enable programmers to work in the language they know and love (well, at least know), in the spirit illustrated in the earlier mention of the EiffelCOM wizard: Generate the IDL or equivalent from the programming language, not the other way around.

All this leads to the idea of a Contract Definition Language (which may take on a different name in its final incarnation). I hope you will agree it's a rather exciting project, addressing what I see as the central issue in component-based development today. Few people would deny that it is at least one of the very top issues. In my next installment, I will present the first design and architecture for such a Contract Definition Language, a design whose appeal will, I hope, be broad enough to be acceptable to many people working with many programming languages, many design methods and many component architectures. ■

ADVERTISER INDEX