Component-Based Development: From Buzz to Spark

The catalyst behind component-based development is the growing realization by the software industry that something must be done to control the costs of developing software products and boosting their quality.

Bertrand Meyer
Interactive Software Engineering/ Monash University

Christine Mingins
Monash University

We software folks easily get carried away by the latest panacea, but there is a difference between a buzz and a spark. Buzzes happen all the time, and their effect lasts anywhere between a few Web-years (defined by convention as traditional trimesters) and a few traditional years. Whether useful, useless, or downright harmful, whether in the end they succeed in affecting actual practice or just fade away, buzzes at best address specific issues. A spark is something else: It happens when the entire industry—or, at least, a substantial group of opinion leaders—suddenly and synchronously "clicks" on an idea and determines that it's going to change forever the way we do our business. Sparks can hardly occur more than once a decade or so. The difference between buzzes and sparks is not value (a buzz can be a great contribution) but scope and depth. Structured programming, Windows, object technology were sparks; many of the fads that have grabbed people's attention in the past few years are just buzzes.

It's still early to tell for sure, but component-based development will most likely become a spark—if we get it right. It is the intent of this special issue of *Computer*, the first on the topic, to help get it right.

Component-based development (CBD) is the building of software systems out of prepackaged generic elements. The current excitement about CBD results from the convergence of four phenomena originating from quite different backgrounds:

- On the scientific side, the progress of modern software engineering ideas with their special emphasis on reuse.
- On the industrial side, the widespread success of theoretically unpretentious but practically useful techniques for building GUIs, databases, and other parts of applications out of components: Microsoft's VBX, OCX, and ActiveX, and Imprise's Delphi. As Wojtek Kozaczynski and Grady Booch wrote in their introduction to *IEEE Software's* Component-Based Software Engineering issue (Sept./Oct. 1998): "Software engineers voted with their keyboards in favor of OLE and Visual Basic."
- On the political side, the push by some of the major players for competing interconnection technologies: CORBA, COM, and Enterprise JavaBeans.
- In the software world at large, the generalization of object technology, which provides both the conceptual basis and the practical tools for building and using components.

The catalyst that has made these four elements fuse into the current CBD movement is the growing realization by the software industry—and more importantly its customers (companies whose business depends on software, external or internal)—that "we can't continue like this forever." Something must be done to control the costs of developing software products and boosting their quality; any solution requires an industrialization of the process, based on the reuse of standard components rather than the still dominant practice of doing too much from scratch.

A fifth element should, in the future, join the above four: Enterprise Resource Planning systems. ERP products (from such vendors as SAP, PeopleSoft, and Oracle) have taken by storm much of the world of management information systems. But they have also caused strong resentment because of their high price, monolithic nature, cost of installation and customization, and general heaviness. Only through componentization can the ERP systems of the future continue to compete; conversely, ERP gives components a chance to affect the very heart of business systems, not just the periphery.

Recent discussions of CBD have focused on binary components made possible by COM and CORBA. There is no reason, however, to take a restrictive view of components—which would mean, in particular, renouncing the benefits of object-oriented libraries developed in OO languages, first Smalltalk and then NextStep, Eiffel, C++, and Java. Object-oriented libraries and frameworks provide components in their own right, and indeed are some of the best ones around. The difference of reuse technology between a class framework and a COM binary affects the form of the components, not the nature of the reuse process.

Quality is the one issue that stands between us and the realization of the CBD objectives.

More generally, in our opinion, it is absurd to present CBD as "the next thing after objects." Not only does object technology pursue the same basic aims as CBD (building software from reusable components), it also provides the only serious known technical basis to achieve these aims. CORBA and COM components, through their interface definition languages, rely on the principles of OO encapsulation; the very word *interface*, as well as the "O" in both acronyms and the very nature of these components (classes), proceed directly from object technology. When it comes to implementing such components, there is really nothing else around other than OO methods, at least if we want to be able to guarantee some degree of quality.

QUALITY

Quality is the one issue that stands between us and the realization of the CBD objectives. It seems not to have dawned yet on the industry that components without a draconian attitude to quality at all stages of the process may be worse than the evils they are trying to cure. It is striking here to see that the literature on CBD—articles in this special issue included—is divided into two almost distinct subsets: those that focus on technical issues and try to take ever more subtle advantage of the available construction and interoperability mechanisms; and those that discuss how to guarantee the quality of the components.

Two of the articles in this issue indeed have quality as their primary focus. In "Making Components Contract Aware," Antoine Beugnard and his colleagues take a closer look at the notion of contract beyond the original correctness contract concept. They distinguish four levels: basic contracts as provided by a simple component interface that lists operations and their signatures (types of inputs and outputs) with no semantic properties; behavioral contracts (the correctness contracts mentioned above) that make it possible to express what operations do, independently of how they do it; synchronization contracts in parallel and distributed systems; and quality-of-service contracts, which bring attention to system-level issues. This is an important classification that will enable component discussions to be based on strong quality concerns.

Cynthia Della Torre Cicalese and Shmuel Rotenstreich's "Behavioral Specification of Distributed Software Component Interfaces" also relies on contracts for describing components' behavioral properties. One of its contributions is a new mechanism, *Biscotti*, for introducing contracts into Java. Numerous Java contract extensions have been proposed, including iContract and Jass. Contrary to most of its predecessors, this one is not preprocessor-based, but a language change implemented through a modification of the Sun JDK source code. Although we think it is preferable to use a language such as Eiffel or Sather with built-in contract support, the article makes a convincing case for the style of extension it proposes. It also contains a cogent presentation of the role of components in distributed systems and the merits of the various proposed approaches.

EXTENDING CURRENT COMPONENT TECHNOLOGIES

An area that is crying out for component-based development is the nec plus ultra of software: operating systems. In "Component-Based APIs for Versioning and Distributed Applications," Robert J. Stets, Galen C. Hunt, and Michael L. Scott, from Microsoft Research and the University of Rochester, present an attractive redesign of the Windows interface based on components. As they are careful to point out, it is not a redesign of the operating system itself, but only of the application programming interface—the system as viewed by application developers through the facilities they can call. They give some convincing arguments for componentizing operating systems: to facilitate OS evolution without endangering legacy applications and to support distributed applications better. Although the article makes no mention of whether and when these ideas will be applied to a commercial operating system—Windows NT being the obvious candidate—one can only hope its advice will be heeded by the designers and maintainers of mainstream offerings.

It is striking to find one of the basic techniques used by Stets and colleagues in the COM world developed again under another name, *interception*, in an article that focuses on CORBA. Interception (which may be viewed as a generalization of the dynamic binding mechanism that plays such a key role in object-oriented development) consists of catching operation requests at the last moment to make them do something different, or something extra, without changing the existing components that carried out the original operations. In "Using Interceptors to Enhance CORBA," Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith discuss the notion of interceptors and its many applications to such areas as security, compression, and scheduling. The range of applications is indeed staggering, especially when extended with the applications described in the previously mentioned COM-oriented article. Here, perhaps more than anywhere else, the differences between the two parts of the component community are striking: If you start invoking component operations at runtime, how do your clients know that you are not violating the operations' original intent? This is one of the areas where the component community will have to come together.

In "Component Assembly for OO Distributed Systems," Guijun Wang, Liz Ungar, and Dan Klawitter from Boeing describe the key architectural concepts that preside over component-based development. They introduce a division of CBD into three levels and a number of fundamental concepts, especially ports and links, which they apply to the study of the principal component standards. This will be particularly useful in focusing future discussions of component-based approaches.

This set of papers offers a rich view of the current trends in CBD. Much more is needed to fulfill the promises of the approach and allow it to have the effect it deserves on the practice of software development. The Component and Object Technology department of *Computer* will continue to present challenging views on the progress in the field; note in particular the first public description of CORBA 3 by Jon Siegel in the May issue, complemented in the present one by Guy Eddon's description of ongoing developments in COM+, Microsoft's strategic effort in the field. Look for more component contributions in the months to come, as the buzz turns into a spark. ❖

Bertrand Meyer is president of Interactive Software Engineering, where he has directed the design of numerous tools, component libraries, and customer applications. His nine books include Object-Oriented Software Construction. *He is the designer of the Eiffel method and language, chairs the TOOLS conference series, and edits* Computer*'s Component and Object Technology department, JOOP's Eiffel column, and Prentice Hall's Component and Object Technology book series. He is an associate member of the French Academy of Science's Application Council and since 1998 has held a position as adjunct professor at Monash University, where he helped start the Trusted Components initiative (http://www.trusted-components.org) as part of the new EDST industry/government consortium for Enterprise Distributed Systems Technology.*

Christine Mingins is associate Head of School in the Faculty of Computing and Information Technology at Monash University (Melbourne), where for several years she has led the use of object-oriented methods at various stages of the curriculum, including introductory programming. Her publications have ranged over software metrics, reusable components, IT education, analysis and design methods, and management issues. She has been for several years the program chair of TOOLS Pacific. She helped start the Trusted Components Initiative and is a member of its core group.

Meyer can be reached at ot-column@eiffel.com and Mingins at cmingins@csse.monash.edu.au.

Get the software that helps you manage risk!

Just how risky is your software project?
Software Engineering Risk Management
Finding your Path through the Jungle
DANGER
Buy *SERIM* and get...
...Software Engineering Risk Management...
...free!
Version 1.0 for Windows 95
member price $180
1999 Codie Award Nominee
A LearnerFirst® Software tool

See our 30-day risk-free guarantee @ http://computer.org

IEEE COMPUTER SOCIETY