

As organizations adopt component-based software engineering, it becomes essential to clearly define its characteristics, advantages, and organizational implications. This report presents key discussion points from a workshop on CBSE and provides a useful synthesis of participants' diverse perspectives and experiences.



The Current State of CBSE

Alan W. Brown, Sterling Software

Kurt C. Wallnau, Software Engineering Institute

Component-based software engineering is generating tremendous interest not just in the software community but in numerous industry sectors. Recent technology advances such as the Web, JavaBeans, ActiveX, and others spur this interest.

But CBSE goes well beyond these technology enablers, as shown by the diverse perspectives brought to a recent workshop on CBSE, held in conjunction with the 20th International Conference on Software Engineering. This diversity also characterized a related ICSE panel discussion, "CBSE: Can it Change the Way of Software Development?" which generated active debate from a large audience. The discussion ranged from the theory of software reuse to the reality of commercial software markets, from available tools to future programming language mechanisms, and from practical testing to rigorous formal specification.

In both the ICSE panel and the CBSE workshop, divergent perspectives at times threatened to blur CBSE's conceptual outlines. Does this diversity imply that we are exploring the same basic CBSE concepts from many different points of view? Or are we exploring fundamentally different or unrelated concepts that we capriciously label CBSE?

WORKING OUT THE DETAILS

On 25-26 April 1998, representatives of industry and academia met in Kyoto, Japan, to participate in an international workshop on component-based software engineering. This workshop, held in conjunction with the 20th International Conference on Software Engineering, aimed to help the software community better understand CBSE and to identify gaps that exist between industry needs and current academic research.

The workshop gave researchers and practitioners an opportunity to share their differing perspectives on CBSE. Synthesizing these helped the participants better understand the nature of components, motivations behind the adoption of CBSE, and potential implications of CBSE on organizations. Although this workshop was not the first or only forum for exploring CBSE concepts, ICSE provided a superb context because it is a premier conference on software engineering.

The workshop comprised a series of panel discussions, organized by the following broad categories:

- ◆ Engineering theory and methodology—what do we mean by CBSE, and how does it influence the way we develop systems?
- ◆ Components and object technology—how does CBSE differ from object technology, and how is object technology used to realize CBSE?
- ◆ Tools and technology—what are the key technologies, how can we use them effectively, and how might we extend or otherwise improve them?
- ◆ Real-life, enterprise-level application of CBSE—why are large enterprises interested in CBSE, what are they doing, and what are their experiences?

In addition to panel discussions, two invited keynote presentations set the tone for each day's session. Mikio Aoyama of NIIT outlined a large-scale vision for CBSE. Wojtek Kozaczynski of SSA detailed the notion of a business component and offered a case study in the use of business components to design a very large-scale enterprise information system.

The workshop focused nominally on component management infrastructures, that is, the build-time and runtime infrastructures for component-based systems. However, as this article shows, the workshop participants charted a more diverse and comprehensive course.

For an online version of the workshop proceedings, go to www.sei.cmu.edu/cbs/icseworkshop.html.

The workshop results suggest the former: CBSE is a coherent engineering practice, and we are making good progress in identifying its core concepts as well as different perspectives on them. We found that such diversity, far from diffusing the concept of CBSE, often works like stereoscopic vision, adding

depth of field to our perception.

For example, while most participants agreed that components act as replacement units in component-based systems, "replacement unit" has different meanings depending on which of two major perspectives is adopted. One, CBSE with off-the-shelf components, views components as commercial off-the-shelf commodities. In this context, CBSE requires industrial standardization on a small number of component frameworks. The other, CBSE with abstract components, views components as application-specific core business assets, which emphasizes component-based design approaches rather than standard component infrastructures or component marketplaces. Although this illustration exaggerates each perspective's tendencies (both influence any large system development effort), it does reflect real differences that conditioned much of the workshop discussion.

This article focuses on the workshop's closing session, which synthesized the major themes from panel discussions. Organized as a facilitated, large-group brainstorming session, it sought to bring into focus the different perspectives on CBSE and identify the major results participants could take away from the panel discussions. The discussion ranged from the conceptual (what is CBSE?) to the skeptical (why will it work now if it didn't before?) to the practical (what will it mean to organizations if CBSE does work?).

WHAT IS CBSE?

Predictably, some discussion focused on definition of terms—notably the term "component." Just as predictably, these definitions only sketched the contours of this complex concept. Fortunately, the workshop had access to several established definitions, and to their credit the participants used these as a basis for exploring additional characteristics of components rather than arguing for or against the validity of any particular definition.

The following definitions of "software component" typify those emerging in the software industry.

1. A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces. (Philippe Krutchen, Rational Software)

2. A runtime software component is a dynamically bindable package of one or more programs

managed as a unit and accessed through documented interfaces that can be discovered at runtime. (Gartner Group)

3. A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition. (Clemens Szyperski, *Component Software*¹)

4. A business component represents the software implementation of an “autonomous” business concept or business process. It consists of the software artifacts necessary to express, implement, and deploy the concept as a reusable element of a larger business system. (Wojtek Kozaczynski, SSA)

These definitions seem to describe approximately the same concept, but close inspection reveals sufficient differences to make them non-substitutable. For example, definitions 1 and (especially) 4 underscore the large-grained nature of components; we might infer this from definition 2, but not from definition 3. Definitions 1 and 3 emphasize the need to accommodate context dependencies, but only definition 3 requires explicit description of context dependencies. We might infer explicit context dependencies from definition 4, but not from definition 2. Similarly, the notion of component autonomy varies—it may refer to a component’s ability to be deployed independently, or execute independently, and so forth.

Each definition has its merits, but rather than debate these in detail, workshop participants noted the importance of two additional component characteristics:

- ♦ the relationship between components and object technology, and
- ♦ the relationship between components and software architecture.

Object technology is neither necessary nor sufficient for CBSE

Curiously, this strong statement—as obvious as it may seem to some—passed without vigorous contention. Instead, workshop participants acknowledged it to be a natural consequence of the panel presentations and discussions—even though most available technologies for component-based development clearly are object-oriented. JavaBeans and Enterprise JavaBeans exemplify component-based technology. The Object Management Group’s

Unified Modeling Language—itsself an outgrowth of object-oriented analysis and object-oriented design—actively addresses component concepts. It therefore seems strange to assert CBSE’s independence from object technology. How can we justify this apparent incongruity?

To state the conclusion first, participants agreed that OT was a useful and convenient starting point for CBSE, but

- ♦ by itself, OT did not express the full range of abstractions needed by CBSE; and
- ♦ it is possible to realize CBSE without employing OT.

Thus, OT is neither necessary nor sufficient. Moreover, CBSE might induce substantial changes in approach to system design, project management, and organizational style—changes that go well beyond those implied by a large and growing base of industry experience with OT.

We see that OT is insufficient for CBSE when we consider the component’s role as replacement unit. The definitions above each address at least one characteristic related to replaceability: explicitly specifying context. Concretely, this might be imple-

CBSE is a coherent engineering practice, but we still haven’t fully identified just what it is.

mented via a “uses” clause on a specification, that is, a declaration of required system resources. This suggestion causes some contention because a “uses” clause implies that the interface describes an implementation rather than an abstraction of possible implementations. OT does not typically support this concept—and there are strong arguments why it should not. However, these lose force when applied to design-level abstractions, especially when attempting to compose using existing components.

To illustrate OT’s non-necessity, we ironically draw on workshop participants’ experiences in attempting to use OT to implement CBSE. Put bluntly, some practitioners are seeking ways to insulate their approaches to CBSE from OT. Why? Because the OT technology market—in particular, distributed OT such as Java, Corba, and ActiveX—is far too unstable and contentious and will, many feel, continue to be so. The workshop discussion tended to treat distributed OT as infrastructure “plumbing” and components as larger-grained abstractions and implementations applicable to diverse infrastructures.

COMPONENTS AND TRANSACTIONS

DAVID CHAPPELL, Chappell & Associates

Business computing depends on transactions. A transaction groups two or more changes into a single unit, then ensures that either all or none of the changes occur. New technologies combine components and transactions, the two most visible being the Microsoft Transaction Server (MTS) and Enterprise JavaBeans (EJB), the latter from a vendor consortium led by Sun Microsystems.

Applications that need support for transactions generally need other services, too; in particular, they often must scale to handle many simultaneous clients. Along with supporting transactions, then, transaction processing (TP) monitors commonly support scalability by allowing effective sharing of resources such as threads, memory, and database connections. To best understand how the advent of components affects TP monitors, let's look at exactly how MTS and EJB provide transaction and scalability services.

To control when a transaction begins and ends, the client may be required to inform the TP monitor of the transaction's boundaries. This is a common solution in non-component-oriented TP monitors, and both MTS and EJB support it. However, MTS and EJB also allow for directly configuring the components' transactional requirements—usually a better solution. It permits using the same component binary in different transactions and facilitates the black-box style of reuse typical of component software. It also frees the client from having to demarcate transactions.

TRANSACTIONS IN MTS

In MTS, every component has a *transaction attribute*. This tells the TP monitor (that is, MTS itself) whether to start a transaction when the first method is invoked in a new instance of this component. MTS defines four possible values for this attribute:

- ◆ *Requires New*: Begin a new transaction upon the first method call to a new component.

- ◆ *Required*: Begin a new transaction only if the component's caller isn't currently part of a transaction. Otherwise, any work this component performs will become part of the caller's existing transaction.

- ◆ *Supported*: MTS never starts a new transaction for this component. If a transaction is in progress for the caller, any work this component performs will become part of it. If no transaction is in progress, this component's work will not be done inside a transaction.

- ◆ *Not Supported*: MTS never starts a new transaction for this component. Even if the caller has a transaction in progress, the new component will not join it—it just doesn't participate in transactions.

To end a transaction, MTS allows a component to explicitly indicate when its work is completed and whether it wants that work to be committed or aborted. To do this, the component calls either `SetComplete` to commit the transaction or `SetAbort`

Continued on the next page

Questioning whether components can be separated from infrastructure led to the following discussion on their relationship.

Components are inseparable from architecture

If one motivation for CBSE is to improve system flexibility through a compositional style of development, we might then ask what makes composition possible. It must exceed our ability to describe abstractions via abstract interfaces—otherwise we would not need CBSE. Instead, the degree to which we are able to “plug in” components—the operative phrase in compositional development—relates directly to the degree to which components adhere to some set of predefined constraints or conventions.

The most prominent component technologies—Enterprise JavaBeans, ActiveX, and Corba (assum-

ing the Object Management Group adopts a component model)—all impose constraints on components. For example, the ability of a component infrastructure to inquire into a component's interfaces requires that the component implement some service or obey some convention as defined by its underlying component infrastructure.

Some participants suggested that components should implement two interface types: a functional one that reflects the component's role in the system, and an extrafunctional one that reflects the component model imposed by some underlying component framework. The latter interface expresses the architectural constraints that enable composability and other desirable properties of component-based systems. Therefore, our understanding of what makes a component a component is inextricably linked to our understanding of the archi-

to abort it. If this component is the only one involved in the transaction, MTS will commit or abort the transaction then. If several MTS components are participating in a transaction, each can call `SetComplete` or `SetAbort` when it completes its work. The transaction doesn't end until the root component—the one created directly by the client—calls either method. If the root and every other component in this transaction call `SetComplete`, the transaction is committed. If any component called `SetAbort`, the transaction is aborted and the work performed by all its components is rolled back.

TRANSACTIONS IN EJB

EJB, like MTS, permits setting a transaction attribute on components. The attribute's primary values—`TX_REQUIRES_NEW`, `TX_REQUIRED`, `TX_SUPPORTS`, or `TX_NOT_SUPPORTED`—closely mirror those defined by MTS. As in MTS, an EJB-based TP monitor uses this transaction attribute to determine whether to begin a new transaction when a method is invoked on a component.

Unlike MTS, every method call a client makes on an EJB component using this style of transaction demarcation is its own transaction. Although this method can invoke other methods in the same or other EJB components, all of which may be part of this transaction, the transaction ends when the client's call returns. EJB does allow a component to call `setRollbackOnly`, analogous to MTS's `SetAbort` call, to roll back the transaction. If any component involved in a transaction invokes this method, work performed by all components will be rolled back. Otherwise, EJB will commit the transaction before returning the method's results to the client.

SUPPORTING SCALABILITY

MTS and EJB define similar services to help developers create scalable applications, including thread management and client authorization. The most interesting distinction, however, is in how each manages a component's state. MTS does not permit a component to maintain its state across a transaction boundary—calling `SetComplete` or `SetAbort` causes that state to be destroyed. This improves resource sharing, since a component can't hang onto anything for very long. It also decreases the possibility that applications will rely on an in-memory state that doesn't match what's in the database after a transaction aborts. But each component must refresh its state after each transaction, which can exact a performance penalty.

EJB, by contrast, allows components to maintain their state across transaction boundaries. This is more natural for developers steeped in the object paradigm, and can avoid the performance penalty of frequently recreating a component's state. However, it also requires that developers diligently ensure effective resource sharing and application correctness.

We're sure to hear much debate about whether MTS or EJB provides better support for creating transactional, component-based applications. Both technologies, however, assertively address the key issues by providing a component-oriented style of transaction demarcation, supporting scalability, and permitting direct management of a component's state.

David Chappell is principal of Chappell & Associates, an education and consulting firm. Readers may contact him at david@chappellassoc.com.

tectural constraints imposed on components by a component framework-cum-object model.

After some discussion, participants decided that although components and architecture seem to go hand in hand, the "two interface" suggestion unduly emphasizes the role of the component framework in software architecture. Indeed, as noted, some participants sought a clean separation between the software architecture and component framework. A more general definition avoids this problem but still preserves a component-architecture duality by recognizing three different views of architecture:

♦ *Runtime.* This includes frameworks and models that provide runtime services for component-based systems.

♦ *Design-time.* This includes the application-specific view of components, such as functional interfaces and component dependencies.

♦ *Compose-time.* This includes all the elements needed to assemble a system from components, including generators and other build-time services; a component framework may provide some of these services.

These additional characteristics that emerged from the discussion suggest that components are complex design-level entities, that is, both abstractions and implementations. Does this complexity help us solve enterprise-level problems? To answer this, we must explore the motivations behind CBSE.

Why CBSE Now?

Over the past decade, many people have attempted to improve software development practices by improving design techniques, developing more

COMPONENT-ORIENTED MIDDLEWARE FOR COMMERCE SYSTEMS

ROGER SESSIONS, ObjectWatch

Writing business logic is hard enough. Writing infrastructure code is much more difficult, and a demanding commerce system requires a complex underlying infrastructure. If you are creating a highly scalable, Internet-enabled commerce system, the infrastructure requirements will be overwhelming.

Modern commerce systems run on four tiers:

- ◆ a Web tier, which manages Web pages;
- ◆ a client tier, to manage computer-human interactions;
- ◆ a component tier, which hosts the business logic packaged as components; and
- ◆ a data tier, to host the back-end databases.

The component tier must handle distributed communications, proxy/instance relationships, security, transactional integrity, communications with legacy applications, data transfer to databases, and scalability, to name just a few. Either you or the underlying component tier infrastructure must solve these problems. The more help you get from the infrastructure, the less work you'll have to do, leaving more time for your business logic. Which probably is complex enough to keep you amused.

So you need a good infrastructure, one focused on components and able to solve component tier problems. Because the component tier is often referred to as the middle tier (between the client and the data tier), we can refer to this infrastructure as *component-oriented middleware*.

THE PRODUCTS

Today's major players in component-oriented middleware are the Object Management Group and Microsoft. Java is trying to enter the fray with Enterprise JavaBeans, but this technology remains premature. The OMG is a technology consortium formed in 1989 by eight companies to define a standard for what they called distributed object applications. It now has over 800 members, and perhaps a dozen vendors implement OMG standards. For the OMG, implementation follows standardization. The OMG does not dictate implementation details except as needed to ensure interoperability, such as the wire format of component requests. Companies can implement the OMG standards as they choose—the theory is that these companies can then compete openly with a variety of implementations that meet various market niches.

Microsoft wants to be the platform of choice for support of component-based applications, which it believes will be the software basis for commerce over the next decade. For Microsoft, standardization follows implementation. Microsoft is a technology company geared to creating competitive implementations. It creates standards only where necessary to allow other companies to plug into its overall component frameworks, such as its two-phase commit framework that allows independent database vendors to participate in Microsoft-coordinated distributed transactions.

Continued on the next page

expressive notations for capturing a system's intended functionality, and encouraging reuse of pre-developed system pieces rather than building from scratch. Each approach has had some notable success in improving the quality, flexibility, and maintainability of application systems, helping many organizations develop complex, mission-critical applications deployed on a wide range of platforms.

Despite this success, any organization developing, deploying, and maintaining large-scale software-intensive systems still faces tremendous problems. Furthermore, in recent years, the requirements, tactics, and expectations of application developers have changed significantly. With this context in mind, workshop participants examined the question "why CBSE now?" and discussed various CBSE solutions.

Two important aspects of the question "why CBSE now?" quickly emerged in the discussion. First, several underlying technologies have matured that

permit building components and assembling applications from sets of those components. Second, the business and organizational context within which applications are developed, deployed, and maintained has changed.

Maturing component technologies

Several participants stated that CBSE is happening now and emphasized that system development methods have changed greatly in the past few years. Development environments such as Visual Basic and languages such as C++ and Java dominate new application development. These languages, and the supporting tools, make it possible to share and distribute application pieces through approaches such as Visual Basic Controls (VBXs), ActiveX controls, class libraries, and JavaBeans. As these technologies have matured, so has understanding about how to develop pieces of applica-

The gulf between hype and reality makes it difficult to compare Microsoft's and the OMG's technologies. The OMG has adopted many standards for which no implementations exist; Microsoft has announced technology that it may not deliver for years. To be fair, then, we should ignore future promises and examine what is generally available and in use today.

For the OMG, "generally available and in use today" means:

- ◆ Corba IDL for defining component interfaces;
- ◆ the basic Corba client–component model;
- ◆ IIOp, the interoperability standard that allows different Corba vendors to work together;
- ◆ Life Cycle Service, to define how component instances are instantiated;
- ◆ Naming Service, to define how component instances are shared;
- ◆ Security Service, to define how clients and component instances work together securely; and
- ◆ Transaction Service, to define how distributed transactions are controlled.

For Microsoft, "generally available and in use today" means:

- ◆ Microsoft IDL for defining component interfaces;
 - ◆ the basic COM client/component model;
 - ◆ DCOM for distributing components across a network;
 - ◆ Microsoft Transaction Service (MTS) to provide a secure runtime environment, transaction management, and scalability;
 - ◆ DTC for distributed transaction coordination; and
 - ◆ Microsoft Message Queue for asynchronous messaging.
- Although OMG seems to be winning the bullet race (seven

to six), Microsoft's MTS product is equivalent to several OMG specifications.

THE RESULTS

Because OMG standards can be implemented on many platforms, customers are not locked into a particular operating system or hardware. However, OMG creates no reference implementations and depends on vendors for actual delivery; this leaves a huge lag between what OMG has standardized and what vendors are actually delivering. It also means that portability between vendors is often more promise than reality.

Microsoft creates implementations; this leaves no doubt about exactly what has been implemented. And Microsoft's control over not only the component-oriented middleware but the underlying operating system permits great efficiencies. On the other hand, component systems based on Microsoft technology cannot easily be ported to other platforms.

If you are designing component-oriented commerce systems, remember that components aren't enough. You also need an infrastructure to support your components. Both the OMG and Microsoft are working hard to convince you that they can and should provide that infrastructure.

Roger Sessions is president of ObjectWatch and consults on distributed component middle-tier technologies. His most recent book is COM and DCOM: Microsoft's Vision for Distributed Objects. He writes the ObjectWatch Newsletter, available at <http://www.objectwatch.com>.

tions following these approaches. Component-oriented development no longer seems foreign to many application developers.

Each of these approaches relies on some underlying services to provide the communication and coordination necessary to piece together applications. The infrastructure acts as the "plumbing" that allows communication among components. To communicate, components must share an understanding of how to use the infrastructure. This could be as simple as a set of naming standards for operations, a standard place to put information about the components, or a conventional way to use other components via the infrastructure (sometimes referred to as a *component model*). This infrastructure may also allow components using the infrastructure to do so effectively and efficiently through services to

- ◆ find out what components are currently connected to the infrastructure,

- ◆ make reference to other components via some meaningful naming scheme,

- ◆ guarantee once-only delivery of messages between components,

- ◆ manage transactions consisting of multiple interactions among components, or

- ◆ allow secure communication between components.

Among the component infrastructure technologies that have been developed, three have become somewhat standardized: the OMG's Corba, Sun's JavaBeans and Enterprise JavaBeans, and Microsoft's Component Object Model and Distributed COM. Participants discussed these infrastructure technologies and related examples of their use in various operational contexts.

Tools and environments supporting each of these technologies are widely available and used, providing many benefits. However, some partici-

pants pointed out the many challenges of using these tools to develop larger applications, manage multiple versions of components, and integrate components developed by different people using different technologies.

Evolving business and organizational context

Far from concentrating exclusively on technology issues, workshop participants broadened the discussion to consider the business and organizational context within which CBSE evolved and must operate, and some important recent developments.

CBSE has broad implications for how we acquire, build, and evolve software systems.

First, the style and architecture of the applications being developed has shifted. Centralized mainframe-based applications accessed via terminals over proprietary networks have given way to distributed, multitiered applications remotely accessible from a variety of client machines over intranets and the Internet. Building such applications requires tools and techniques that support new development methods and approaches. Organizations that once handled a few large projects now typically manage many smaller projects whose results must be shared.

Second, organizations have invested significant financial and intellectual resources in the applications they have built over the past two decades. This has left fewer resources for developing new applications from scratch, making it essential to leverage and reuse the existing investment across a range of operations and in developing new applications quickly and reliably. To achieve this, developers require greater support and guidance for decomposing applications into meaningful pieces and for assembling new applications from a mixture of new and existing pieces.

Third, organizations began to realize the strategic implications for their business practices of the software-intensive systems that support their organization. Some found themselves locked into proprietary software solutions, at significant cost. Typically this arose from two sources.

Those who attempted to develop large parts of their software infrastructure (both systems and application software) on their own often found themselves responsible for a growing—and very expensive—software maintenance backlog. This put them at a disadvantage against more agile organizations

that could quickly respond to customer and market changes by updating or replacing their computer infrastructure.

Other organizations found that they had relied too much on a single product from a single vendor. The resultant “vendor lock-in” made it difficult to take advantage of a free market of computing suppliers, left important decisions about computing infrastructure in the hands of third parties, and often significantly reduced the ease with which information could be shared among partner organizations. Complete, packaged applications proved difficult to customize readily as specific organizational needs arose.

Organizations therefore began to look for an appropriate balance, an approach that did not require writing everything from scratch each time, and that provided a flexible system that could evolve as needed to meet changing business needs.

Finally, and perhaps most importantly, the business environment in which organizations operate has changed drastically. To succeed, an organization must maintain some stability and predictability in its market, in the technology supporting its core businesses, and in its internal structure. Unfortunately, the rate of change in all these areas rises inexorably, making the ability to manage complexity and adapt rapidly to change an important differentiator among competitors.

The solution to these problems seemed to lie in a software development approach that addresses each of these requirements. As stated in the workshop, the goals of CBSE include the ability to

- ◆ embrace opportunities offered by new technologies in software system delivery and deployment;
- ◆ encourage reuse of core functionality across applications;
- ◆ enable flexible upgrade and replacement of system pieces whether developed in-house, supplied by third parties, or purchased off-the-shelf; and
- ◆ encapsulate organizational best practices such that they can be adapted as business conditions change.

CBSE AT WORK

Workshop participants experienced with CBSE development projects were quick to point out that CBSE involves much more than simply using object request brokers, setting up a library of useful code, or acquiring Visual Basic controls over the Internet.

These tactical approaches must be supplemented with strategic thinking and planning for successful adoption of CBSE. In particular, CBSE has broad implications for building, acquiring, assembling, and evolving systems, raising some important concerns.

We can distinguish two classes of concerns based on whether components are

- ♦ used as a design philosophy independent from any concern for reusing existing components, or
 - ♦ seen as off-the-shelf building blocks used to design and implement a component-based system.
- For this discussion, we denote these as CBSE with abstract components and CBSE with off-the-shelf components, respectively.

CBSE with abstract components

This approach involves radically rethinking the relationship between design, requirements, and components. Fundamentally, it requires new methods for software development, new processes, and powerful tools to automate generation and management of components and interfaces. These CBSE-oriented methods and tools, currently under development, provide an interface-based design focus that concentrates on a solution's basic component architecture.

In participants' experience, this approach stabilizes system design at the interface level, concentrates attention on collaborations among interfaces as the basis for understanding a system architecture, and enables reuse and replacement of implementations that conform to the interface specifications. Several emerging CBSE methods²⁻⁴ are being tracked or applied in current projects, but participants emphasized the need for further experience with these.

Some pointed out one important consequence of this revolution in design approaches: a dramatic change in software engineers' primary roles and required skills. Some organizations moving toward CBSE find they must rethink how they organize teams to concentrate on component provisioning within a well-defined component architecture. Finding people who can operate in this environment is proving to be a major challenge, and the lack of appropriate skills within an organization could severely hamper CBSE's adoption.

Wojtek Kozaczynski pointed out the major day-to-day challenge for organizations moving to CBSE: managing component-based applications as they are deployed, and maintained, and continue to evolve. Multiple components will provide similar functionality, many versions of the same compo-

nent will emerge, multiple configurations of component sets will be in use, and so on.

Traditional configuration management and version control techniques provide an important starting point to manage some of these issues. As monolithic development and deployment approaches yield to component-oriented methods, however, new management methods and tools will prove essential. In particular, high composeability in a product line setting amounts to mass customization, which introduces tremendous configuration management challenges and support challenges. Many opportunities for new tools and techniques exist in this area.

CBSE with off-the-shelf components

This approach moves organizations from application development to application assembly. Constructing an application now involves the use of predeveloped pieces, perhaps developed at different times, by different groups of people, and with many different uses in mind. One scenario highlighted in detail at the workshop illustrates the implications of such a change: black-box assembly of commercial off-the-shelf components.

An organization assembling COTS components to create a command-and-control application for the US government found it had limited access to the components' internal design, predefined options for customizing the components' behavior, no ability to influence the release cycle of new component versions, and total reliance on the long-term viability, integrity, and ability of the packages' maintainers. This affected many aspects of the design, assembly, testing, deployment, and maintenance of the system.

In such cases, the development effort becomes one of gradual discovery about the components, their capabilities, their internal assumptions, and the incompatibilities that arise when they are used in concert.⁵ As happened in this example, the architecture of a COTS-based system often degenerates into a series of contingency and risk mitigation strategies based on this discovered information. The workshop concluded that a growing emphasis on outsourcing of systems and increased use of COTS components will demand many improvements in how such components are documented, assembled, adapted, and customized.

We can expect dramatic change in engineers' primary roles and required skills.

Does your library get IEEE Software?
Would you like your colleagues to read a copy other than yours?
Complete this card and give it to your librarian!

ATTENTION LIBRARIAN/DEPARTMENT HEAD: I would like to recommend IEEE Software for acquisition.

Name _____
Dept _____
Date _____
Signature _____

Sample copies are available from
IEEE Computer Society
PO Box 3014, 10662 Los Vaqueros Circle
Los Alamitos, CA 90720-1314 FAX 1.714.821.4010
ISSN 0740-7459 Bimonthly: \$430
Or ask about package rates.

As software technology becomes a core part of business enterprises in all market sectors, customers demand more flexible enterprise systems. This demand coincides with a maturing software technology infrastructure for building distributed enterprise systems. CBSE is a new style of software system development emerging from this growing demand and maturing technology. While CBSE is still evolving, and perspectives vary on what it is all about, there is little doubt that something is happening, that we are calling that something CBSE, and that its outlines are becoming clearer all the time. ❖

ACKNOWLEDGMENTS

We thank the ICSE workshop organizers, and Mikio Aoyama in particular, for the excellent workshop arrangements. Thanks also to Wojtek Kozaczynski for his detailed recording of the summary session, and to Wojtek Kozaczynski, Philippe Kruchten, Chris Dellarocas, David Carney, and Mikio Aoyama for their comments on this summary.

REFERENCES

1. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison Wesley Longman, Reading, Mass., 1998.
2. D. D'Souza and A.C. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison Wesley Longman, Reading, Mass., 1998.
3. P. Coad, *Java Design: Building Better Applications and Applets*, Prentice Hall, Upper Saddle River, N.J., 1997.
4. P. Allen and S. Frost, *Component Based Development for Enterprise Systems: Applying the Select Approach*, Cambridge University Press, New York, 1997.
5. D. Garlan et al., "Architectural Mismatch: Why Reuse Is So Hard," *IEEE Software*, Nov. 1995, pp. 17-26.

About the Authors



Alan W. Brown is Director of Research for Sterling Software's Applications Development Division. His responsibilities include coordinating research activities across the organization and advising on the future of component-based software development products. Prior to this, Alan spent five years at the Software Engineering Institute at Carnegie Mellon University, where he led the CASE Environments Project that advised US Government agencies and contractors on the application and integration of CASE technologies.

Brown's primary research interests include component-based development, software engineering environments, and CASE tools. Among his many publications in these areas are Principles of CASE Tool Integration, Object-Oriented Databases and their Application to Software Engineering, and Software Engineering Environments. He obtained a BSc in computational science from University of Hull and a PhD in computing science from University of Newcastle upon Tyne.



Kurt C. Wallnau is a senior technical staff member at the Software Engineering Institute (SEI), where he leads the COTS-Based Systems project. His current interests include the role of product evaluation in system design and the movement from COTS-based to component-based systems. Prior to joining SEI, Wallnau was System Architect of a DoD program focused on designing systems for the use of COTS software.

Wallnau received a BS in computer science from Villanova University.

Address questions about this article to Brown at Sterling Software, Applications Development Division, 5800 Tennyson Parkway, MS/142, Plano, TX 75024; Alan_Brown@sterling.com.