

Component-Based Systems: A Classification of Issues

Although component-based development offers many potential benefits, such as greater reuse and a commodity-oriented perspective of software, it also raises several issues that developers need to consider.

Pearl
Brereton
David
Budgen
Keele
University

Developing and using various component forms as building blocks can significantly enhance software-based system development and use,¹ which is why both the academic and commercial sectors have shown interest in component-based software development. Indeed, much effort has been devoted to defining and describing the terms and concepts involved. Briefly, we describe software components as units of independent production, acquisition, and deployment that interact to form a functional system.² See the “What Is a Component?” sidebar for a detailed definition and description of a component.

In this article, we identify a set of issues organized within an overall framework that software developers must address for component-based systems (CBS) to achieve their full potential. Participants in the 1999 International Workshop on Component-Based Software Engineering (<http://www.sei.cmu.edu/cbs/icse99/cbsewkshp.html>, Aug. 1999) developed a framework similar to ours, which helps validate our model.

In component-based development, although significant difficulties can arise from the inexact notion of a component, maintaining at least a semi-vague notion of a component can be valuable. Doing so helps avoid a so-called “technology lock-in,” which narrows the scope of our thinking. Components can take a wide range of forms and sizes; they should be independent of specific software architectural style; while objects may be components, all components are not objects.³ Therefore, our framework leads to a more effective understanding of components because it helps clarify those aspects of the component concept that are largely

independent of architectural and implementation issues.

Classifying and grouping the relevant ideas into a framework achieves the following:

- *Helps transfer the potential of component-based system development into reality.* Although still immature, the concept of component-based development is timely. The necessary constructional technologies are available to support and exploit it, and it offers potential gains in productivity and reliability.
- *Brings together disparate perspectives on components.* Thinking about components incorporates a wide range of views, from those that reflect primarily business objectives⁴ to those that are almost entirely concerned with technical issues. We believe that these views should all be complementary parts of a larger whole, and we have therefore sought to bring them together within a single framework.
- *Begins to identify the key research questions.* Much as was the case when they began adopting object-oriented technology, developers currently believe that component-based development offers great potential, but have yet to translate that potential into practice. However, one lesson OO taught us is that we must identify where bottlenecks will likely occur so that we can address them at an early stage.⁵

DERIVATION OF THE FRAMEWORK

Our framework resulted from extensive discussion and debate within a research group composed of aca-

demics, postgraduate students, and undergraduate summer interns who perform software engineering research. One motivation for producing a components framework was to provide a vehicle for applying our research to component-based development.

We created the framework in three phases, as Table 1 shows. In phase one, we studied the issues and the published material. In phase two, we brainstormed extensively to identify relevant issues and found that we needed to address these issues from several viewpoints. We initially established a 3×3 framework of issues versus viewpoints. The framework considered process, product, and people issues from the view-

points of component providers, component integrators, and component-based system users.

Using our framework to elaborate our ideas in phase three revealed that the initial classification of process, product, and people was insufficient. We then subdivided people issues into business issues and people in software development. Further review revealed that many issues spanned either two or all three viewpoints. This breadth of relevance proved particularly noticeable for business issues, which may suggest that many of these nontechnical issues likely concern a whole range of future component-based-system stakeholders.

What Is a Component?

Stuart Thomason, Keele University

There are at least as many definitions of a component as there are readers of this article. Perhaps you have chosen to read this because you are working with components or because you have an interest in them. In presenting our framework, we have tried to be as universal as possible, so whatever your definition of a component is, you will find some relevance here. As a baseline, however, we agree with Alan Brown and Keith Short¹ that a component is “an independently deliverable set of reusable services.”

Independence does not necessarily mean that a component has no dependencies on other components, although such a characteristic is often desirable, merely that those dependencies are generic enough for several different providers to satisfy. For example, a word processor can depend on a spell-checking component, but several perhaps rival products could meet this need. The word processor can be supplied independently, provided its dependencies are adequately specified. This situation gives component-based system integrators and end users more flexibility and choice.

Alan Brown and Kurt Wallnau analyze more restrictive definitions from both industry and academia.² The concept of the explicit interface dominates these definitions. Although the inner workings of a component can be treated as a black box, its interface must be explicitly defined. This includes a complete listing of the services provided and how to access them, generic

dependencies, and error conditions that might occur. Many definitions include either build-time or runtime dynamism. As individual units of change, components can be swapped, replaced, or upgraded during the build of a complete application. More complex component architectures allow component services to be looked up and hooked up while an application runs.

The scale and size of components vary enormously—there are no real limits—but readers familiar with component-based software engineering will agree that, in general, a single component provides insufficient functionality for forming a complete system. A fundamental tenet of CBSE is that components provide services that can be integrated into larger, complete applications.

The Component Space

Any component can exhibit varying degrees of distribution, modularity, and independence of platform or language; we can thus map components onto a three-dimensional space.

Within this space, we place monolithic systems at $[0,0,0]$. That is, they are nondistributed, nonmodular, language dependent, and platform dependent. Visual Basic components, in their standard form, are neither distributed nor language independent, placing them at $[0,1,0]$. Contrary to popular opinion, both CORBA and Java fail to achieve the nirvana of a $[1,1,1]$ placing. CORBA is

distributed and language independent, but the underlying components often remain platform dependent. Similarly, while Java components are cross-platform in scope, they are generally language specific. However, wrapping a platform-independent language such as Java within language-independent middleware, such as CORBA, would yield a component worthy of $[1,1,1]$ status.

We have used a binary scale for simplicity, but recognize that some technologies are more distributed than others. We also acknowledge that our three-dimensional space should be populated using a continuous rather than a discrete scale.

References

1. A.W. Brown and K. Short, “On Components and Objects: The Foundations of Component Based Development,” *Proc. 5th Int’l Symp. Assessment of Software Tools and Technologies*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 112-121.
2. A.W. Brown and K.C. Wallnau, “The Current State of CBSE,” *IEEE Software*, Sept./Oct. 1998, pp. 37-46.

Stuart Thomason is a research fellow at Keele University working on a European Union-funded project involving the development of a software component broker. He recently completed a PhD in the configuration management of components.

Table 1. Phases in framework development.

Phase	Focus	Method
One	Background knowledge sharing. Familiarity with terminology, issues, and activities in component-based systems research.	Selection and presentation of several key publications on component-based systems. Presentation of our individual research from a component perspective.
Two	Identification of key issues in process, product, and people.	We brainstormed and listed each issue on a copy board.
Three	Population of a 3 × 3 framework, identifying issues from the viewpoints of component providers, component integrators, and component-based-systems users.	We brainstormed and recorded further issues on sticky notes, discussing and moving them within the framework.

SOFTWARE PRODUCT ISSUES

Typically, software engineers view product issues primarily from the perspectives of component providers and component integrators. The two groups also share some common needs. Customers are also concerned with product issues, particularly how to specify requirements effectively, but they focus more on commercial issues, as opposed to production-oriented concerns.

Component providers

Two important factors component providers must consider are component *granularity* and *portability*.

Granularity. Components come in all shapes and sizes. However, the level of granularity or size has significant consequences for the provider, affecting

- the choice of development tools and languages,
- production time scales, and
- the investment level required for unit and integration testing.

Providers must measure and estimate granularity to position themselves within the component supply chain. Conventional measures of size and complexity—such as lines of code and function points—have significant limitations and, in particular, fail to accommodate factors associated with reuse.

One proposed approach to describing a component's size that could address these limitations of more conventional measures is to count the number of use cases a component supports.⁶ This measure has yet to be validated.

Portability. This factor relates to both transferring software between systems and executing software within a range of operating environments. Component providers express particular concern with the latter, which considers how much a component depends on other software (including software architecture), hardware, and its operational context. Also important are the degree of *visibility*—whether the dependence is implicit or explicit—and the dependence level.

The use of middleware standards, such as the Common Object Request Broker Architecture (CORBA) and Microsoft's Component Object Model/Distributed Component Object Model (COM/DCOM), and platform-independent languages such as Java partially address portability issues.

Component integrators

Four product issues of particular relevance to component integrators are component selection, component interoperability, quality control of integrated components, and integrated-systems maintenance.

Component selection. Identifying and evaluating available options is a principal task of component integrators. To make trade-offs among components, architectures, and requirements, integrators must consider

- how to characterize requirements,
- how to characterize components, and
- mechanisms for finding potentially useful components and evaluating them against requirements.

Component characteristics of interest to integrators include both technical factors (such as functional capabilities, behavior, and nonfunctional attributes or constraints) and commercial considerations (such as cost, availability, and supplier reputation). A wide range of appropriate approaches to capturing, representing, and measuring these characteristics will likely emerge, so selecting and combining the best approaches will be a problem in itself.

The reuse community has done considerable work to address the problems associated with the representation, storage, and retrieval of reusable modules and components. The methods for accomplishing this include keyword, faceted, attribute-value, and enumerated classifications.⁷

Interoperability. Integration can range from straightforward and possibly automated fabrication to a complex set of activities that involve gluing, wrapping, or adapting components. Potential difficulties include

- *architecture mismatch*, which occurs when components fail to meet the architectural constraints;
- *functional deficiencies*, which arise when components do not satisfy all the functional requirements; and
- *quality maintenance*, which involves somehow combining quality attributes when developers integrate several components in a system.

Component adaptation, or the use of special-purpose wrapping to achieve interoperability, will likely

significantly increase subsequent maintenance costs for integrated systems because developers must readapt or reglue components as they evolve.

Restricting the component types used for a particular integrated system—for example, using *only* JavaBeans—can significantly reduce interoperability problems. The limited types of components in the component marketplace reflect the current popularity of this approach.⁸

Combining quality attributes. Developers must also address how the integrated system inherits properties associated with product parts. For example, if you integrate several high-performance or high-reliability components, what can you say about the performance or reliability of the system as a whole? Similarly, if you integrate a combination of low- and high-quality components, how can you assess and improve the resulting system's quality?

Maintenance. Similar to Web documents,⁹ the responsibility for and possibly the location of system components is distributed across multiple organizations, complicating the evolution and maintenance of a component-based system. This distribution significantly complicates maintenance tasks for testing, evaluation, and ultimately distribution, including locating and evaluating potential replacement components and determining the impact of potential evolutionary paths and system building. In particular, from a production perspective, maintainers of component-based systems need methods and tools to help them understand components and configurations and carry out regression testing of new system versions.

Standards play a significant role in reducing maintenance problems. However, we must ensure that standards use does not lead to a lack of choice and reduced opportunity for innovation.

Common needs

We identified two product issues—predicting limits and component description—as being of particular importance to both component providers and integrators.

Predicting limits. Components of an integrated system are likely to be developed within very different organizational structures and software engineering environments. They may also be implemented in different languages. Therefore, underlying assumptions relating the validity of operations at the boundaries of operand values must be explicit. For example, a component may specify a word length of 32 bits but not be able to perform valid operations with extreme values within this range.

We can perhaps best view this issue as one of portability at a lower level of abstraction than described earlier.

Component description. Developers must describe com-

ponents so that integrators can locate, understand, and evaluate them. These integrator activities will, to some extent, be automated; therefore, developers must describe components in a way that is suitable for use by automated tools and is also understandable to human integrators.

Developers also need to address the problems of accumulating and combining information that contributes to component descriptions. Such information will likely come from several sources and vary in form, being factual, opinion based, or statistical. Types of information and sources might include

- advertising literature and component introspection from component suppliers;
- interface standard compliance certifications from standards bodies and certification organizations;
- component reliability measures from other users such as those in special-interest groups; and
- assessments based on benchmarking, case studies, and experiments from integrator organizations.

Researchers and developers have proposed and are using several approaches to component description.

Integrated systems customers

Customers should specify requirements in a way that makes it possible for integrators to identify and undertake the trade-offs necessary in selecting suitable components. Clearly, the gap between the requirements specification and component description languages will affect an integrator's ability to achieve a satisfactory outcome. The closer the match between the languages, the easier it should be to provide the best solution.

To support the trade-off process, requirements specifications should incorporate information about a particular requirement's importance—whether it is mandatory, high priority, or optional—and the degree to which the customer can tolerate differences between component capabilities and system needs.

SOFTWARE DEVELOPMENT PROCESS ISSUES

Development process issues can affect just one viewpoint or all viewpoints.

Component providers

Providers must address internationalization and testing-practices issues.

Internationalization. Software's increasingly global nature suggests the need for a development process that supports internationalized software. In such a process, developers could easily parameterize components for

Developers must describe components in a way that is suitable for use by automated tools and is also understandable to human integrators.

Integrators must attempt to control and manage the disparate and potentially overwhelming demands for change.

different languages and cultures, although such flexibility risks introducing some degree of redundancy. In addition to accommodating a range of natural languages—for user interfaces and error messages, for example—developers also must account for cultural variations.

One example of such a variation is in the meaning of the ✓ and ✕ symbols. Anglo-American cultures generally interpret these symbols as opposites, whereas, in most of Europe, the two are often interchangeable. Software development globalization could help address this problem.¹⁰

Testing practices. A second issue for component providers is the need to develop suitable context-independent component testing practices. Where practices cannot achieve such independence, component providers should make any dependency explicit. Context independence is an important element in making components widely acceptable because developers are notoriously and understandably apt to be wary of any code produced by others, even within a single organization.

Component integrators

As might be expected, component integrators will face some of the most significant challenges. Although component integration is nothing new, it is undergoing a significant shift in process. The widespread integration of commercial off-the-shelf products and in-house components will likely shift the focus of software engineering from the “specify, design, and implement” concept toward “select, evaluate, and integrate.” The issues addressed here are trade-offs between requirements and the capabilities of available components, tool support, and demands for change.

Requirements and component capabilities trade-offs. Over the past three decades, development practices have implicitly been based upon the idea of building anew each time. So the first challenge is to develop new approaches for developing systems that will incorporate concepts such as goodness of fit so that developers can make trade-offs when choosing between adapting their solution to use what is available and developing new components.

Several process models now incorporate reuse, although not specifically commercial off-the-shelf component reuse. Some noteworthy examples of in-house reuse in practice have also been documented. However, component integration within a commercial environment is significantly more than reuse, and the area needs further investigation and experimentation to identify appropriate process models.

David Carney and Kurt Wallnau make a useful contribution to this debate and argue that “COTS software evaluation is an inherently diverse topic, involving

unavoidable uncertainty and a range of technical and business analysis techniques that span well beyond what is traditionally thought of as evaluation.”¹¹

Tool support. Any new procedure requires the appropriate tools. Reusing components involves new activities such as

- searching for components and determining how to identify approximate fits,
- comprehending what components do,
- evaluating the consequences of each available choice, and
- visualizing the effects of each decision.

Support for these activities must also focus on packaging and reuse, rather than development and evolution.

Demands for change. Integrators must also attempt to control and manage the disparate and potentially overwhelming demands for change. These changes include push activities, as vendors produce new offerings that integrators must track and evaluate. It also includes pull activities, such as customer demands for new features. Some of the latter could also be generated internally by the need to identify new markets and products (meeting “design-to-order” demands and making speculative developments).

Accommodating change requires different practices than those employed to produce custom systems or packaged systems such as word processors and spreadsheets.

Common needs

As Figure 1 shows, all three viewpoints address two process issues: long-term support and the responsibility chain.

Long-term support. The potential need for long-term support is likely to be a major factor in purchasing components because history suggests that even throw-away systems can remain in use for a long time. Employing a mechanism such as escrow—which stores source code at an independent, trusted, and secure repository—may help reassure customers. Practices such as employing multiple sources, as with hardware systems, and using preferred suppliers could provide further reassurance.

Responsibility chain. The second common process issue likewise relates to confidence and concerns the responsibility chain that arises when multiple providers combine to construct the final product. The nature of software makes it notoriously difficult to extract the source of a particular fault, even when the elements are produced locally. For the producer-integrator-customer model to operate effectively, developers must address this problem in a way that’s acceptable to all three viewpoints.

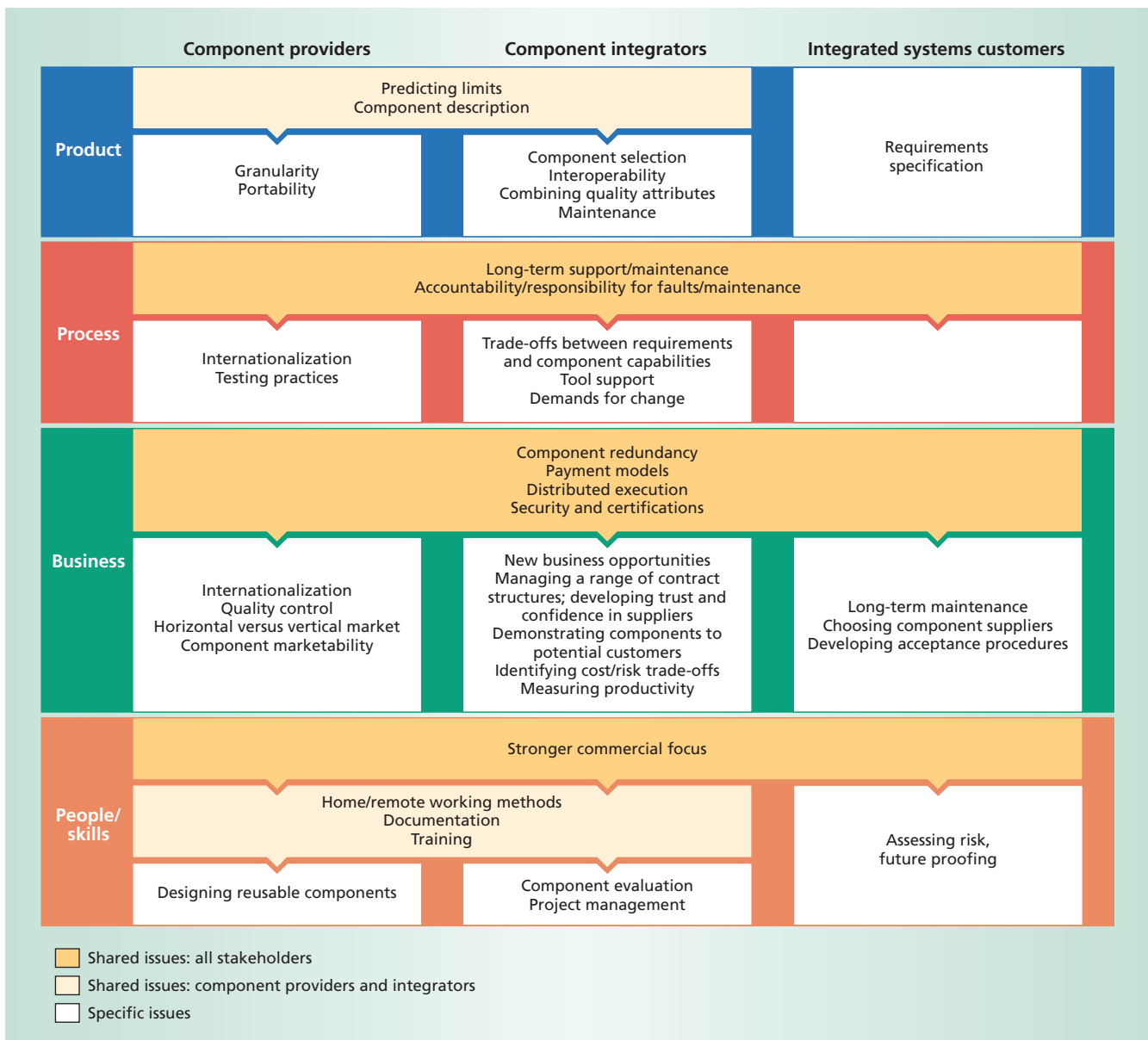


Figure 1. Key issues, stakeholders, and a framework for component-based systems. Each column represents one stakeholder group: component providers, component integrators, or integrated systems customers. Each row shows a functional area of the framework: Software product, software process, business, or people/skills. Within each functional area, issues that concern a single group appear in that group's column. Issues that concern multiple groups span two or more columns.

BUSINESS ISSUES

Similar to software development process issues, the commercial angle involves issues important to a single viewpoint, as well as issues that concern all stakeholders.

Component providers

The issues that are of concern from the component provider's perspective are mirrored to varying degrees

within the component integrator viewpoint.

Internationalization. Software components are just one example of a commodity that can be bought and sold in a global electronic marketplace. As such, they must adhere to legal requirements within the countries of use, both in terms of operational characteristics, such as encryption regulations, and marketing practices, such as advertising regulations.

Component providers need to make their components as appealing as possible to a broad range of integrated products and product domains.

Responsibility for quality. Components providers, especially COTS components providers, can exert little control over how others use their components. They must, therefore, limit their level of responsibility for components in use and for the integrated products of which the components are a part. One approach to addressing this problem is to adopt the practice of service-level agreements between component suppliers and procurers.

Horizontal versus vertical market. There may be markets for components both within domains (vertical market) and across domains (horizontal market). It seems probable that generic horizontal components will provide value at the user interface and at lower infrastructure levels. Clearly, business components are more likely to succeed within a particular domain, but identifying and attracting cross-domain interest may lead to greater market share.

horizontal components will provide value at the user interface and at lower infrastructure levels. Clearly, business components are more likely to succeed within a particular domain, but identifying and attracting cross-domain interest may lead to greater market share.

Marketability. As with any commodity, component providers must consider the marketability of components. In particular, it is important to straightforwardly communicate strengths and features and to build in obsolescence so that opportunities for selling new versions of components continue to arise. Component providers need to avoid dependence on a few integrated product lines, making their components as appealing as possible to a broad range of integrated products and product domains.

Component integrators

Component integrators have a strong commercial focus. In particular, an organization that assumes this role is concerned with

- identifying new business opportunities;
- managing a range of contractual styles with both suppliers and customers;
- building relationships with trusted suppliers;
- demonstrating products to potential buyers;
- managing trade-offs between the capabilities of known components, costs of evaluating new components, and system requirements; and
- measuring productivity.

New business opportunities. The widespread availability of cheap components provides integrators with significant business opportunities. On the one hand, such organizations may be large and relatively stable, providing well-supported products to business customers. At the other extreme, integrators may be small, dynamic, entrepreneurial businesses that provide services to individuals who seek lifestyle accessories in a wide range of as yet untapped domains.

Managing a range of contractual styles. Integrators clearly need to establish new contractual structures that

bridge the gap between component providers and customers of integrated products. They must inevitably deal with several supplier and customer organizations that are subject to different national rules and regulations. The overhead associated with large and complex supply chains could result in the emergence of preferred suppliers that integrators trust, as is evident in other component-based industries such as car manufacturing.

Demonstrating products to potential buyers. Integrators need to demonstrate integrated products to customers as part of their marketing activities. Component pricing models must accommodate this product demonstration. Essentially, integrators should probably develop and formalize the current and growing practice of acquiring or accessing software for a trial period.

Trade-offs. Earlier, we identified the need for integrators to address trade-offs between requirements and available components within the development process. From a commercial perspective, several related issues arise. For example, integrators need to consider the relative costs and risks involved in

- evaluating many components,
- accepting a component that's nearly good enough, and
- building an ideal component from scratch.

It is, of course, also important to decide whether or by how much integrators can tolerate relaxing requirements while remaining commercially viable.

Measuring productivity. Traditional methods measure software engineering productivity based on quantity of code produced—the number of lines of code. Clearly, this approach is unsuited to an environment that practices large-scale reuse. Integrators must develop new productivity models and apply them to component integration.

Integrated systems customers

Ideally, customers should see only benefits such as improved quality, wider availability, and reduced costs. In practice, however, they may also see a major limitation in terms of support for product failures and product evolution.

From a customer's point of view, overcoming the problems associated with maintaining component-based systems can be a critical business requirement. The need for reliable product evolution may lead to a growth in organizations that provide third-party maintenance, an escrow facility for source code, or both. Customers could reduce maintenance problems by purchasing systems that use only preferred suppliers or by implementing more stringent acceptance procedures. Customers could also require integrators to take responsibility and liability for system evolution and support.

Common needs

As with process issues, some business issues are of concern to all stakeholders.

Component redundancy. Vendors now commonly sell software products with built-in extra features. Following the initial purchase, customers pay more to gain access to extra product features. This practice can clearly be applied to component-based systems. However, integrated products may become unacceptably large, which could make controlling and charging for a disparate set of evolution options extremely complex.

Payment. Billing may become a major overhead for the component-based software industry. If, for example, customers pay for components on a per-use basis—where such use may involve remote execution—payment models might need to incorporate payment to both component providers and systems integrators. The industry would then need a much broader range of future payment models to accommodate both the complex webs of owners and agents and the different purchasing and licensing models.

Distributed execution. So far, we have assumed that components are physically integrated to provide an executable system that runs on a customer's computer. However, current technology supports the development of virtual integrated systems in which components remain at the provider sites, or at some other remote site, and systems access them as required. Such systems could reduce problems associated with system upgrades and version management. However, widespread distributed execution could bring its own difficulties in terms of system performance, a dependency on the underlying communications infrastructure, and the security of commercial information.

Security and certification. Security and certification will clearly be major issues in component-based system development and use. Integrators will play a significant role in virus checking, trust and risk assessment, and assessing the benefits and overhead of using certificated components. As with any off-the-shelf package or component, there will be a trade-off between implied quality through certification and time to market.

Independent evaluation, certification bodies, or both could provide a test quality rating that would act as a component's dependability score.¹²

PEOPLE IN SOFTWARE DEVELOPMENT

Many of the issues we identify suggest that the development and use of component-based systems will significantly affect both software engineers and customers. Both development teams and customer organizations will need new skills. People issues are either of particular concern to one of the viewpoints or they span two or more viewpoints.

Component providers

The activities and problems associated with component development have much in common with those arising in the development of traditional software-based systems. Designers, however, must focus more strongly on designing reusable systems that can be easily and safely integrated with other components. Despite being a topic of concern to designers for many years, reuse has yet to exert a significant impact on design practices.

Component integrators

Evaluation and management are areas that will particularly need new skills. In addition, integrators are most likely to be successful if they are both committed to reuse and also have confidence in systems produced by developers within other organizations.

Evaluation. A principal task of a component integrator is to evaluate and compare components. This evaluation is a fundamental part of making the necessary trade-offs between component cost, component availability, component functionality, component quality, and integrated-system requirements. Integrators need a significantly greater awareness of evaluation methods and practices than is currently evident.

Management. Managers of component integration projects need to be highly skilled in both the technical and commercial aspects of product development. There is already a shortage of people with these dual skills. Clearly, industry and government must invest significantly in education and training to ensure the availability of enough high-quality component integration project managers to meet future demands.

Integrated systems customers

Customers concerned about long-term system maintenance need to assess the risk of available options. Risk analysis skills are already in place in large organizations and those projects where safety issues are critical, such as military procurement. This is not the case, however, in smaller organizations and less safety-critical domains.

Common needs

Component providers and integrators have several issues in common. In particular, they should anticipate an increase in both work-at-home and distributed-work methods. Technologies to support these work styles are now mature, and a component-based approach to software production could accommodate software life cycles with less dependence on co-located developer teams.

Providers and integrators also need to have a greater focus on documentation, and they may need to acquire new skills in document production, integration, and maintenance.

Despite being a topic of concern to designers for many years, reuse has yet to exert a significant impact on design practices.

Across all viewpoints, personnel need to develop a stronger business focus, including a greater awareness of the economic, legal, and ethical consequences of their actions.

The component-based systems approach could potentially overcome many difficulties associated with developing and maintaining monolithic software applications. In particular, the approach should result in better quality products, more rapid development, and an increased capability to accommodate change.

One reason for developing a framework was to help identify key research questions. Several themes have emerged as important areas requiring further research if component-based system development is to become established software engineering practice:

- *Evaluation* includes the assessment of components, component suppliers, and integrated component-based systems; development and maintenance strategies and alternatives; and architectures, risk, productivity, and skills.
- *Maintenance* includes issues relating to responsibility; ownership; component and system comprehension; managing disparate demands for change, risks, trust, and confidence; evaluating evolution options; using preferred suppliers; payment options; and documentation.
- *Interaction and integration of commercial and technical factors* relates to the selection of components and suppliers; trade-offs among cost, functionality, quality, and availability; future proofing; managing change; payment models; and commercial and technical process integration.
- *Aggregation rules* for component properties such as performance, reliability, and level of certification when developers combine components to form integrated systems.

Underpinning each of these themes, as well as many of the issues we've examined, is the need for appropriate and explicit component description and documentation. Research in this area should address the range of description forms, sources of descriptive information, description maintenance, users and usage of descriptions, and description quality. *

Acknowledgments

We thank Barry Cook, Neil White, Ka Po Lam, Stuart Thomason, Amnart Pohthong, Adel Taweel, and Keith Pugh for their useful contributions to discussions about the issues identified here. We also thank the referees for their constructive comments and suggestions for improvements.

References

1. O.P. Brereton et al., "The Future of Software: Defining the Research Agenda," *Comm. ACM*, Dec. 1999, pp. 78-84.
2. C. Szyperski, *Component Software—Beyond Object-Oriented Programming*, Addison Wesley Longman, Reading, Mass., 1998.
3. A.W. Brown and K.C. Wallnau, "The Current State of CBSE," *IEEE Software*, Sept./Oct. 1998, pp. 37-46.
4. D. Andrews, ed., *Butler Report: Component-Based Development*, 1998, Report Series, Vol. 1, Butler Consulting Group Limited, Hessele, UK, 1998.
5. R.G. Fichman and C.F. Kemerer, "Object Technology and Reuse: Lessons from Early Adopters," *Computer*, Oct. 1997, pp. 47-59.
6. M. Tsagias and B.A. Kitchenham, "An Evaluation of the Business Object Approach to Software Development," to be published in *J. Systems and Software*, Nov. 2000.
7. W.B. Frakes and T.P. Pole, "An Empirical Study of Representation Methods for Reusable Components," *IEEE Trans. Software Eng.*, Aug. 1994, pp. 617-630.
8. V. Traas and J. van Hillegersberg, "The Software Component Market on the Internet: Current Status and Conditions for Growth," *Software Eng. Notes*, Jan. 2000, pp. 114-117.
9. O.P. Brereton, D. Budgen, and G. Hamilton, "Hypertext: The Next Maintenance Mountain," *Computer*, Dec. 1998, pp. 49-55.
10. J.Z. Gao et al, "Engineering on the Internet for Global Software Production," *Computer*, May 1999, pp. 38-47.
11. D.J. Carney and K. Wallnau, "A Basis for Evaluation of Commercial Software," *Information and Software Technology*, Dec. 1998, pp. 851-860.
12. J. Voas and J. Payne, "Dependability Certification of Software Components," to be published in *J. Systems and Software*, Nov. 2000.

Pearl Brereton is a reader at Keele University, Staffordshire, England. Her research interests include software configuration management, component-based systems, and distributed collaborative working. Brereton received a PhD in numerical analysis from Keele University. Contact her at o.p.brereton@cs.keele.ac.uk.

David Budgen is a professor of software engineering at Keele University. His research interests include design, measurement, and evaluation practices for software-based systems. Budgen received a PhD in theoretical physics from the University of Durham. Contact him at db@cs.keele.ac.uk.