Component Based Software Engineering: A Broad Based Model is Needed

Allen Parrish (parrish@cs.ua.edu)
Brandon Dixon (dixon@cs.ua.edu)
David Hale (dhale@alston.cba.ua.edu)

Department of Computer Science
Area of Management Information Systems
The University of Alabama
Tuscaloosa, AL 35487

February 1999

Abstract

Over the past few years, a number of different models of component-based software engineering have been proposed and discussed. We argue that many of these models are too narrow and/or informal, and that a broader and more precise foundation is needed for CBSE.

1. INTRODUCTION

Component-based software engineering (CBSE) has existed in one form or another for a number of years. The idea of constructing modular software has long been recognized as advantageous within the software community, even dating back to the early days of FORTRAN programming with subroutines and libraries serving as "components." Work by Booch [2] and Meyer [5] in the 1980's was generally regarded as seminal in the advancement of ideas regarding the fundamental nature of components, particularly with regard to low-level structural properties of the components. More recent work [1,7,8,10,12] extended these ideas along various dimensions, including the introduction of formal specifications into component frameworks, the development of new paradigms for data movement, and the development of improved design guidelines for what constitutes a good component that is both efficient and independently verifiable.

Over the past few years, advances in enterprise computing and client-server communities have generated renewed interest in the concept of CBSE, bringing the terms "component" and "component engineering" into widespread use within the software community. (Publications such as *Component Strategies* and the recent CBSE special issue of *IEEE Software* [3] represent the current popular view.) However, most current popular references to these terms are in a much different context than that described above. In particular, current popular CBSE references are to technologies such as COM, CORBA and JavaBeans that support the encapsulation of so-called "binary" components. Such CBSE references and the associated technologies are also linked to the support of distributed object computing, where the notion of a component seems to be linked (perhaps equivalent to) an encapsulated object-oriented software unit that is deployed within a distributed architecture.

We feel that there is a tacit impression given in much of the current dialog that the modern notion of components is based upon a brand new set of ideas and concepts. For example, the

following quote appears in [4]: "Components are now an evolution beyond objects, incorporating all the best aspects of objects, adding important new engineering concepts such as separation of interface and implementation, and enabling easier development through provision of rich runtime services." The ideas listed here (e.g., separation of interface and implementation) have been an important part of object-oriented development from the very beginning. A slightly different quote appears in [11]: "Objects and components come from the same family. Like siblings, they squabble about their differences, but have much more in common than they are willing to admit." This quote implies a popular view that places objects and components in totally distinct categories, a view that this author is trying to dispell. However, the fact that they are in completely different categories to begin with is disturbing, as we believe that the relationships are strong and obvious.

The current popular view of components appears to be summed up by Szyperski [9]: "Software components are binary units of independent production, acquisition and deployment that interact to form a functioning system." In particular, the fact that software components are *binary* units appears to be widely assumed in much of the current popular dialog. We do not object to this view *per se*, but we simply claim that it is inappropriate as a *starting point* for proving a foundation for CBSE. Such a starting point seems to suggest that CBSE originated with the advent of binary component technologies (e.g., COM), and that previous work in the areas of source code components and object-oriented technologies is only weakly related to the modern notion of CBSE. In contrast, our position is that the foundation of CBSE should rest on a generic model of components that allows basic CBSE concepts to be expressed as broadly as possible, and results in the field to be applied as broadly as possible. If such a model is designed properly, then constraints can be added at the appropriate time to obtain more specialized notions of components.

2. A FUNDAMENTALS PERSPECTIVE

We propose that the underlying definition of component be approached as a kind of "conceptual theory," by proposing a generic, formal definition of component to use as a starting point. As an example of the beginning of such a theory, we say that a *component* is a software artifact consisting of three parts: a *service interface*, a *client interface* and an *implementation*. Roughly speaking, the service interface consists of the services that the component exports to the rest of the world; the client interface consists of those services used by this component exported from other components, and the implementation is the code necessary for the component to execute its intended functionality. To enforce the idea that a component must interact with other *software*, we might also want to include a property that the service interface *or* the client interface might be empty, but not both.

To borrow from the electronics domain, we say that a service interface consists of one or more *receptacles*, while a client interface consists of one or more *connections.* A connection is *plug compatible* with a particular receptacle if there is formal consistency between the two. The nature of the formal consistency depends on the types of components and types of interfaces.

As with any theory, there are necessarily a number of undefined terms. The idea though is to permit any realization satisfying the formal structure to be treated as a component model. For example, a simple library of C functions may be viewed as a component under this model. In this

case, the service interface is the collection of function prototypes, the client interface is the set of external functions called by functions in the library, and the implementation is the aggregate of all of the function implementations.

Other component realizations of our model include:

- A C++ class.
- A cluster of C++ classes, with a particular class serving as the exported interface, and the other classes functioning as part of the implementation.
- A Windows DLL.
- A COM object.
- A Java Bean.
- A CORBA-based server object.
- A Unix shell program (i.e., functioning within a pipe-and-filter style architecture).

In each case, it is a straightforward process to identify and justify all of the elements in our generic model. Effectively, our model seems to treat almost any software artifact with one or more identifiable interfaces as a component. This may appear to suggest that our model is too generic to be of any practical use. But it should be recognized that we are only proposing a possible "lowest-level" foundation for CBSE. That is, our first principles-based argument is that software modules with connections to other software modules are, at some level of abstraction, components. In the most general sense, all such modules share some very general properties. Thus, when thinking at this level, every time we invent some new kind of software artifact, we are not necessarily inventing something *totally* new. In our opinion, this is a very important concept to clarify.

Once this generic foundation has been established, restrictions may be added. For example, one could simply restrict the universe of components to those components in binary form. Indeed, it might even be appropriate to argue that the "component" label should only be applied to binary artifacts, as is the current popular usage of the term. If this were the case, by *starting* with the less restrictive model and then adding the binary requirement, the strength of the connection between the restricted notion of components and other non-binary artifacts (such as object-oriented classes) would be more explicit. In this case, we might even formally refer to the more general definition as a "module" and the restricted form as "component." Thus, "component" is a type of "module", sharing some generic properties with "modules" that are *not* "components." By identifying what these properties are, the inherited legacies from other technologies are more explicit, and it may be easier to distinguish truly new concepts from reincarnations of old ones.

This model is definitely work in progress. Our objective was to simply outline a possible direction for the solution of an extremely difficult problem. We believe that an incremental approach, where minimal, first-principles properties are first identified and then restrictions are progressively added, is the only way to arrive at a clear, unambiguous foundation for what CBSE should be about. We also note that this view is similar to a recent article by Betrand Meyer [6], who believes in a strong linkage between object-oriented technology and components, and proposes various dimensions for characterizing components. We believe that an extension of his framework into a full-scale, multi-dimensional characterization is badly needed (perhaps

supported by a generic, minimalist foundation such as the one described here).

REFERENCES

[1] Batory, D. and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 4, October 1992, pp. 355-398.

[2] Booch, G., *Software Components with Ada* , Benjamin-Cummings, 1987.

[3] Brown, A. and K. Walinau, "The Current State of CBSE," *IEEE Software*, vol. 15, no. October 1998, pp.37-46.

[4] Kora, D. "Build vs. Buy – Maximizing the Potential of Components," *Component Strategies*, vol. 1, no. 1, July 1998, pp. 22-35.

[5] Meyer, B., *Reusable Software: The Base Object-Oriented Component Libraries*, Prentice-Hall International, 1994.

[6] Meyer, B., "On to Components," *Computer*, vol. 32, no. 1, January 1999, pp. 139-140.

[7] Sitaraman, M. and B. Weide, eds., "Special Feature: Component-Based Software Using RESOLVE," *Software Engineering Notes* , vol. 19, no. 4, October 1994, pp. 21-67.

[8] Weide, B., W. Ogden and S. Zweben, "Reusable Software Components," *Advances in Computers* , vol. 33, M.C. Yovits, ed., Academic Press, 1991, pp. 1-65.

[9] Szyperski, C., *Component Software: Beyond Object-Oriented Programming*, Addison Wesley, 1998.

[10] Smargdakis, Y. and D. Batory, "Implementing Reusable OO Components," *Proceedings of the International Conference on Software Reuse* , June 1998.

[11] Taylor, D., "Are Objects Obsolete?", *Component Strategies*, vol. 1, no. 1, July 1998, pp. 16- 17.

[12] Zweben, S., S. Edwards, B. Weide, and J. Hollingsworth, "The Effects of Layering and Encapsulation on Software Development Cost and Quality," *IEEE Transactions on Software Engineering*, vol. 21, no. 3, pp. 200-208.