A Component Model Proposal

Jim Q. Ning

Andersen Consulting

3773 Willow Road

Northbrook, IL 60062, U.S.A.

+1 847 714 2537

jning@cstar.ac.com

Abstract

This position paper describes a conceptual model for Component-Based Software Engineering (CBSE). The model is an attempt to define what CBSE is essentially about and help answer critical questions concerning how CBSE relates to and distinguishes itself from other software development paradigms/concepts such as object-orientation.

1. Motivation

As clearly stated in the opening statement of this year's CBSE Workshop's theme description:

> "There is growing interest in the notion of software development through the planned integration of pre-existing software components. This is often called component-based development (CBD), component-based software engineering (CBSE), or simply componentware. While the broad concepts of CBSE are well known and easily stated, a closer look reveals that the term CBSE is used in a diverse set of situations, encompasses a variety of characteristics, and is often given many different interpretations."

To clarify the misconception and confusion, this position paper proposes a model or framework which, hopefully, will be simple and easy to explain to people on one hand and yet rich enough to capture all the key component-related concepts. In the next section, a concept diagram is presented to depict the key concepts and their relationships. Then, a glossary is provided to further define/describe the concepts. Finally, I will explain why such a component model is important and beneficial to the CBSE community.

During the workshop, I anticipate to receive comments from the participants to improve the proposed model. The objective is for this young community to reach some consensus on what CBSE is essentially about and help answer critical questions concerning how CBSE relates to and distinguishes itself from other software development disciplines such as object-orientation.

2. Concept Diagram

The concept diagram is shown on the last page of this paper (Appendix). It is meant to illustrate the important CBSE concepts, the relationships among themselves and with other software engineering concepts. The component concepts are shown in plain (white) boxes, and other concepts in gray boxes. The following link types are used to describe the relationships between the concepts:

- **Aggregation**. This is shown as a solid-line path with a hollow diamond at one end. The concept that is connected by the diamond end is the aggregate.

- **Association**. A binary association is shown as a solid-line path that connects two concepts. A ternary

association is shown as a diamond with a path from the diamond to each participant concept. The cardinality of a participant concept is optionally shown at the end of an association path.

- **Generalization**. This is shown as a solid-line path from the more specific concept to the more general one, with a hollow triangle at the end where the path meets the general concept.

The two vertical dotted lines in the diagram group the concepts into *component concepts* (those in the left column), *interface concepts* (those in the middle column), and *connector concepts* (those in the right column), the three cornerstones of CBSE. The two horizontal dotted lines further categorize the concepts in terms of when they are created during the software lifecycle: the what I call *Type concepts* at the top row are developed at the design/specification time, *the Instance concepts* at the bottom row are created at runtime, and the concepts in the middle row are typically generated during system construction. More precise definitions of these concepts can be found in the next section. It should be noted that no Type concept equivalent is shown for a component concept. It is my belief that a component type at the specification level will be too abstract to be useful.

The gray boxes around the borders of the diagram show how more conventional software artifacts relate to component concepts. Note that this diagram is not meant to be exhaustive. The gray boxes are selectively drawn to exemplify what the component concepts are (e.g., a Component Instance is an Executable), what they are composed of (e.g., a Class is a part of a Component), how they interact with other concepts (e.g., a Connector Instance receives services from an Object Request Broker), etc.

3. Glossary

This section provides definitions/descriptions of the component concepts shown in the concept diagram.

- **Component** – An encapsulated, distributable, and executable piece of software that provides and receives services through well-defined interfaces.

- **Component Instance** – The runtime manifestation of a component. It is typically a runtime image or a piece of code run within some runtime image.

- **Interface Type** – An abstract specification of a set of behaviors without the concern of how to implement the behaviors.

- **Interface** – The association of an interface type to a component to make the services provided or required by the component externally visible.

- **Provided Interface** – An interface representing the services supported by a component to the external world.

- **Required Interface** – An interface representing the services that must be received by a component from outside in order for the component to perform its own operations.

- **Interface Instance** – The runtime manifestation of a interface. It is typically the proxy, stub, and marshaling code packaged and run within some component instances.

- **Connector Type** – An abstract specification of a style of interaction among components.

- **Connector** – The association of a connector type between the required and provided interfaces of components.

- **Connector Instance** – The runtime manifestation of a connector. It can be an independent runtime image or be packaged as part of the proxy and stub code within some component instances.

4. Benefits

What can we gain by having a component model that will be commonly accepted by the CBSE community? First, we

will have a common framework to talk about components in particular and CBSE in general. How many times have we seen a meeting or workshop getting into endless discussions on what is and is not a component? Amazingly, however, no consensus has been reached after all these discussions – a sign of an emerging but immature discipline. The outcome of this work will hopefully become a starting point for the community to brainstorm on and evolve with to build a common understanding and a solid foundation for CBSE.

Another benefit of having a well-defined component model is to help distinguish component concepts from other related concepts. For example, using the concept diagram given in this paper, we can easily state that a program module may form part of a component implementation. But a module by itself is not a component - a module in the conventional sense does not include explicit descriptions of provided and required interfaces. Similarly, we can decide that an object in the object orientation sense is not a component either. An object can be an identifiable entity that lives within some component instance at runtime. But it does not fit our general definition of a component.

Yet another benefit is that such a model helps us understand how the component concepts exist within the context of known technology pieces. For example, the diagram tells us that the concept of interfaces is not entirely new, ports and application programming interfaces (APIs) are all special cases of component interfaces. The diagram also shows that a connector instance does not exist on its own; it must be supported by some runtime infrastructure services such as transaction monitors.

There is no question that a good component model is long overdue for the CBSE community.

References

1. Bronsard, F., Bryan, D., Kozaczynski, W., Liongosari, E., Ning, J., Ólafsson, Á, and Wetterstrand, J., "Toward Software Plug-and-Play," in *Proceedings of the Symposium on Software Reusability*, 1997.

1. D'souza, D. F., and Wills, A. C., *Objects, Components, and Frameworks with UML – The Catalysis*^{TM} *Approach*, Addison-Wesley, 1998.

2. Hofmeister, C., Nord, R., and Dikip Soni, *Applied Software Architecture – Draft*, 1998.

3. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 1995.

1. Rational Software, *UML Notation Guide*, Version 1.1, September 1997.

1. Shaw, M., and Garlan, D., *Software Architecture – Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

Appendix – Component Concept Diagram