From Component Infrastructure To Component-Based Development

Alan W. Brown

Sterling Software
6620 Chase Oaks Blvd., M/S 8515
Plano, TX 75023, USA.

Alan_Brown@sterling.com

1. Challenges to CBD

Much of the existing work in **component-based software technology** has concentrated on developing infrastructure capabilities and middleware solutions for connecting independent pieces of system functionality. The result of this work is a range of maturing infrastructure products for supporting deployment of distributed systems. Examples include message-oriented middleware (MOM) products such as Microsoft's MQ Server and IBM's MQSeries, and object request brokers (ORBs) such as Microsoft's Distributed Component Object Model (DCOM) and the various implementations of the Object Management Group's (OMG's) Common Object Request Broker Architecture (CORBA) [3].

The availability of these products has led many application developers to consider their use in the development and deployment of large, distributed, mission-critical applications requiring robust operation in the face of high transaction rates, multiple simultaneous users, and so on. However, to achieve this requires substantially more than the component infrastructure products now available. It requires a move from component infrastructure products to a broader notion of **component-based development (CBD)**. In particular, to be effective, developers of large-scale, mission-critical applications require many additional capabilities, including ways to:

- reengineer legacy applications to harvest existing components reusable in other applications, or replaceable by newer technologies;
- find suitable components both locally and externally;
- integrate components implemented in a variety of different technologies;
- validate a component's behavior before using it;
- manage multiple implementations of the same component in different technologies, and as it evolves over time.

It is our belief that large-scale adoption of component-oriented approaches in these demanding business applications must be built on four key advances:

- *methods* for designing CBD solutions that help the organization focus on the major functional pieces of their domain, and how those pieces will interact;
- *tools* that support specification of business components using techniques that allow the functionality to be described independently of a particular implementation technology;
- *implementation techniques* for components that support demanding requirements for key business goals such as performance, usability, availability, etc.;
- a *component management and assembly infrastructure* to knit together all the pieces specified, built, and acquired, even when those pieces have been developed using different technologies, or by different people.

The next two sections of this paper briefly expand on these ideas, and focus on how these advances can be realized in practice. The paper concludes with a brief summary.[1]

2. CBD-Oriented Methods

The move toward CBD requires existing software analysis and design approaches to be reconsidered. In particular, any method supporting CBD is required to exhibit at least the following 3 key principles:

- *A clear separation of component specification from its design and implementation*. This allows component behavior to be described independently of its implementation, and supports the possibility of multiple alternate component implementations for the same specification.
- *An interface-focused design approach*. The goal of the CBD method must be the definition of encapsulated behavior accessible through well-defined interfaces. Interfaces provide a contract between providers and consumers of services allowing greater independence across components.

[1] We only address the first 2 issues due to space limitations. A much fuller treatment of these ideas is available elsewhere (e.g., [2]).

- *More formally recorded component semantics*. Informal descriptions of component behavior can be provided via operation signatures and informal text. However, details of operation semantics require formal, verifiable descriptions using preconditions and post-conditions attached to each operation. Without this, component behavior is ambiguous and open to misinterpretation by its potential users.
- *A rigorously recorded refinement process*. Stages in the specification and design process must be recorded to maintain a design record of a component's evolution. Justification for each refinement must also be captured. Use of a component by a third party requires this level of information to assure its quality and to understand aspects of the designers' rationale.

To satisfy these requirements, we consider an approach to component modeling inspired by Catalysis, a "next generation methodology for modeling and constructing open systems from components and frameworks" developed by Icon and TriReme [1].

There are seven key ideas that characterize a Catalysis-inspired approach to component modeling and satisfy the requirements described above. We discuss each of these in turn.

2.1 Describe the static aspects of the domain

A user describes the static structure of the elements of interest within a domain as a set of related types in a type model. The structural relationships among types represent the static constraints that exist among elements of the domain.

For each type in a domain the user describes its features (attributes and operations) in detail. Particularly important are the pre and post conditions that define the semantics of each operation by describing the state that must exist before the operation can take place, and the state that will result having executed the operation. Informal definitions of the pre and post condition can be given. However, more valuable are pre and post conditions in some formal, verifiable notation supported by the component modeling tool.

2.2 Describe the dynamic aspects of the domain

Interactions among types are modeled as collaborations. Changes of state in a domain occur through interactions among behavior bearing types in that domain. These interactions are represented as collaborations in which types play roles in which they initiate or respond to requests to carry out actions.

A collaboration diagram records the interactions among types in the domain as a sequence of messages (operation invocations). A type responds to a message by invoking the named operation with the given parameters and performing the state change defined in the pre and post conditions of that operation.

2.3 Allow multiple views of the domain

At any time the user may wish to focus attention on some set of types or interactions in a domain. To do this a user must be able to create views focussed only on those elements that are of interest for some specific purpose.

2.4 Track each important step in the design process

As a user progresses with their modeling there will be important stages in which they will wish to record the current model. This can be used for backup purposes should the user wish to return to that point in the modeling, and as an historical record of design rationale of how the final model evolved.

Modeling progression within a domain is recorded through the concept of model conformance. A conformance is a relationship between two descriptions of the same thing (types, collaborations etc.) [1]. A conformance is accompanied by a mapping that justifies the conformance claim. Several types of conformance exist, including: A component implementation conforms to the specification; A class that implements a set of behavior conforms to a type that specifies the behavior; A set of fine-grained actions conform to a more abstract single joint action.

2.5 Support reuse of previously modeled (parts of) domains

It is typical for modeling to be carried out by groups of users over an extended period. To support this it must be possible to model discrete parts of a domain, and allow those parts to be combined in semantically-meaningful ways. Domains can be considered to act as scoping boundaries for describing behavior. A user can import one domain into another, or can decompose one larger domain into a number of smaller domains. This supports both top down and bottom up development methods.

2.6 Package appropriate behavior as interfaces of a component

Having modeled the static and dynamic aspects of a domain, the user must decide how that behavior should be packaged in terms of implementable units which may be developed independently, shared across projects, and executed on different machines.

This packaging takes place by describing which interfaces will be packaged within a single component specification. Each behavior-bearing type is an interface offering a set of operations. The user selectively decides on the grouping of those interfaces into components specifications. Each component specification is an identification of the interfaces it supports.

2.7 Check for completeness and consistency

A user may perform component modeling purely as an intellectual exercise to provide greater understanding of some area of their business. However, more typically a user performs that modeling as the step toward one of two goals.

The first goal is to create a component specification which can serve as a definition of the requirements for some externally acquirable or acquired implementation. In this case the user will not implement the component themselves, but will rely on the component specification as a complete and unambiguous contract to be met by an external provider of that implementation.

The second goal is to create a component specification as a preparatory step before producing an implementation satisfying that behavior. In this case the user will either directly implement each of the operations offered by the component in some programming language, or will develop models describing the implementation details of those operations from which code can be automatically generated for some target platform. In either case the component specification is the basis on which the implementation will be created.

In a component modeling tool there must be a number of consistency and completeness checks which the user can execute to ensure that the component specification is suitable for either of these goals.

3. Tool Support for CBD

Designing and building components requires new tools and techniques. With middleware and the underlying component technologies beginning to mature, we can now concentrate on identifying other elements in a CBD tool architecture.

A CBD tool architecture should of course itself be component based. The architecture should allow users to identify and use their preferred components. Thus a CBD tool architecture is actually a framework. Such a framework however, needs to define the basic technology within which the selected components will operate.

Figure 1 shows an example of an architecture for a CBD toolset that illustrates many of the basic elements required for CBD support. In particular, the figure shows the CBD toolset divided into its three tiers of client, rules, and server. The client aspects provide the functionality made available to users of the toolset. It consists of modeling, rendering, model management, and implementation services. The rules aspects provide the underlying component modeling engine supporting a CBD approach. The server aspects provide persistent storage and interchange of component models with external data sources.

One of fundamental beliefs with respect to CBD is that a CBD toolset must provide the capability of *generating* technology-specific components using the component specification data recorded in the information model of the component modeling engine, and persisted in the some form of data repository. Although the primary implementation environment is the generation component, it is sometimes necessary to transform component model information to external generation environments. Transforming information from one model to another is conceptually an easy problem. It only becomes difficult when we try to minimize the information loss during the mapping process. Transform tools are necessarily hand written for each target implementation environment.

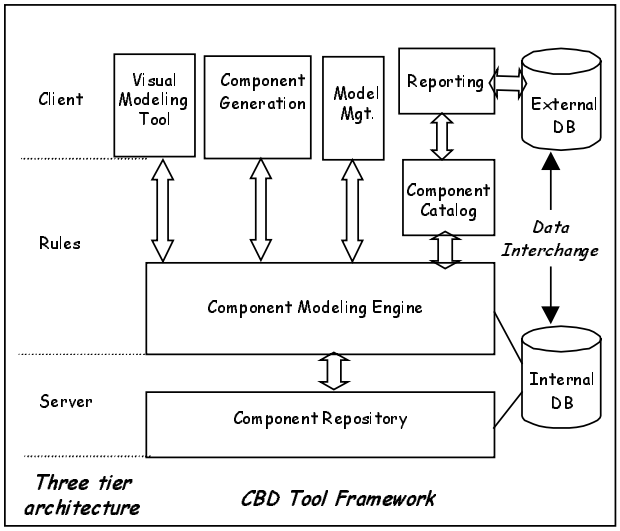

Figure 1: An Illustrative CBD Tool Architecture.

In the remainder of this section we focus on one of the key elements of this CBD tool architecture: the persistent data repository. The repository is a central element in this illustrative tool architecture. Many possible products could be used for this element, but perhaps the Microsoft Repository offers the greatest promise of sophisticated functionality, wide industry acceptance, and compatibility with an extensive range of tools. More specifically, for a tool developer such as Sterling Software, the Microsoft Repository is important for three main reasons.

First, Sterling Software's participation in the joint design effort of the Microsoft Repository led to the production of several public information models to enable the storage of components in the Microsoft Repository. Hence, this choice is an obvious one for us, and relates well to existing products with which we are familiar.

Second, the information models utilize a subset of the Unified Modeling Language (UML) information model

while extending the model with other component modeling concepts. More specifically, the Component Description information model (Cde) address the areas of component specification, implementation and executable information. The mapping from component modeling approaches to this information model is therefore well-defined and straightforward.

Third, a standard information model means that many organizations can offer components to be stored in the Microsoft Repository or use components currently stored in the Microsoft Repository. Its ubiquitous nature (currently it is packaged as part of the Visual Basic product) suggests that it is likely to become the dominant standard for storing component information. Figure 2 illustrates how such a repository can be used to store and use components.

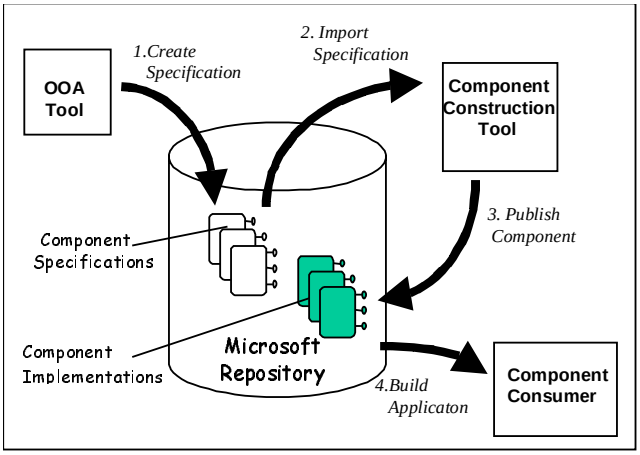

Figure 2: A Component Repository.

As the use of the Microsoft Repository flourishes, the types of components stored and the types of applications being constructed will become apparent and the need for a component cataloging facility will become paramount. Although the existing component information model provides some rudimentary constructs for component searching, more sophisticated modeling will be required. A component catalog engine which facilitates sophisticated search and retrieval can be appended to the component modeling engine. The user interface for the component catalog can utilize the user's preferred Web Browser.

A CBD tool architecture as described above provides several advantages. Users can select their preferred UML compliant OOA tool. For leading OOA tools the architecture will provide standard mappings of concepts in the OOA tool to Component Modeling concepts allowing for some limited generation of the necessary transform tools. Standardization on the underlying tool technology (e.g. COM, Automation) also means that the necessary knowledge to thread together a preferred set of tools is widely available.

4. Summary and Conclusions

Component-based development of software is an important development approach for software systems which must be rapidly assembled, take advantage of the latest web-based technologies, and be amenable to change as both the technology and application needs evolve. One of the key challenges facing software engineers is to make CBD an efficient and effective practice which does not succumb to the shortcomings of previous reuse-based efforts of the 1970s and 1980s. The keys to this include:

- Separation of component specification from component implementation to enable technology-independent application design;
- Use of more rigorous descriptions of component behaviors via methods that encourage interface-level design;
- Flexible tool architectures for CBD based on existing tool technologies and standards.

This paper has briefly explored some of these issues in the context of CBD tool support that will enhance the effectiveness and viability of large-scale software reuse through the sharing of components – software packages offering services through their interfaces. The result is an interface-based approach to application development and design that encourages the creation of systems that are more easily distributed, re-partitioned, and reused.

The ideas expressed in this paper are being pursued by Sterling Software and form the basis for a new generation of CBD technology that will greatly improve the ease with which component-based systems can be developed, deployed, and upgraded.

Acknowledgments

This paper draws on the work of Icon Computing and TriReme in developing the Catalysis method. For further information on Catalysis see http://www.iconcomp.com.

The ideas and techniques described in this paper have been developed cooperatively with a number of colleagues at Sterling Software. In particular, we recognize the important contributions made by Balbir Barn, John Cheesman, John Dodd, Bill Gibson, Paul Sanders, and Keith Short.

References

1. Desmond F. D'Souza and Alan C. Wills; "*Objects, Components, And Frameworks with UML – the Catalysis Approach*"; Addison-Wesley, Reading, Mass. 1997.
2. Balbir Barn and Alan W. Brown, "*Improving Software Reuse Through Component-Based Development Tools*", Submitted for Publication, October 1997.
3. A.W. Brown (Ed.), "*Component-Based Software Engineering*", IEEE Computer Soc. Press, Los Alamitos CA, 1996.