11

COMPONENT-BASED SOFTWARE ENGINEERING

Software component development seeks to develop a set of reusable software components that satisfy particular asset specifications. This is generally done either by developing the components from scratch or through "reuse-based" reengineering of legacy artifacts. Most of the current work in component-based development is focused on infrastructure development and the accompanying middleware to connect independent components to provide system-level functionality.

Component-based software engineering (CBSE) is an approach to application software development in which prefabricated, pretested, and reusable pieces of software are assembled together to form flexible applications that are built rapidly. A good metaphor for CBSE is children's Lego blocks. CBSE is sometimes referred to as component-based software development, but this term omits the critical element of component design and specification, which must precede the development.

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." Early work on components was performed by Grady Booch (1987) for Ada 83 and Bertrand Meyer (1997) for Eiffel.

CBSE is drawing a lot of interest because of the emergence of enabling architectures and SDKs such as ActiveX, JavaBeans, .NET, and the Web. But CBSE goes beyond just technology. Suppose an application designer could draw on a pool of software objects whose form and function were as obvious as Lego blocks. Then

Software Paradigms, By Stephen H. Kaisler
ISBN 0-471-48347-8 Copyright © 2005 John Wiley & Sons, Inc.

applications would be as easy to build as "snapping" together the software modules. An organization could move from *application* development to *application* assembly. Unfortunately, it is not so simple.

11.1 DEFINING CBSE

Component-oriented programming aims to replace traditional monolithic software systems with reusable software components and layered component frameworks (Szyperski 1998). The frameworks provide an execution environment for plugging in components developed by third parties. These components must conform to the interfaces specified by the framework and implement or extend framework functionality.

CBSE is a coherent engineering methodology: it incorporates both a process and a technology description. The process, emphasizing reusability, includes system design, project management, and system life cycle concerns. The technology description, based on some of the definitions of components given in Chapter 8, enables the creation of applications.

CBSE is based on three principles:

- *Separate Component Specification from Its Design and Implementation:* This allows independent implementation of the component, including multiple implementations that can support a plug-and-play architectural approach.
- *Focus on Interface Design:* Interfaces provide a contract between clients and servers that allows encapsulated behavior that is independent of other components.
- *Formal Specification of Component Semantics:* Associated with the interface specification is a semantics specification that helps to guide the implementer in developing the component.

While CBSE has often been linked with object-oriented programming (OOP), OOP is not a prerequisite for practicing CBSE. We can build components using programming languages that do not support object-oriented behavior as long as we define an appropriate methodology for how these components are designed and used.

11.2 PROBLEMS WITH CBSE

CBSE requires much work before we can rise to the level of flexibility and ease of use associated with Lego blocks. Five problems must be addressed:

- Building components.
- Finding components.
- Using components.

- Testing components.
- Harvesting components for re-use.

11.2.1 Building Components

How does a designer build a reusable component? This is, perhaps, the thorniest problem facing CBSE. What is reusable to one designer is an obstacle to another.

A key question is whether software can be designed in isolation of its intended use. For low-level components, the answer is fairly obvious. The C++ Standard Template Library and Booch's Ada components are good examples of reusable components. But these components are data structure components rather than functional components for a specific domain.

Let's look at another aspect of the problem. Suppose we could design reusable components. How would we describe them in such a way that potential clients could find them, determine that they are useful, and assess whether they would cause any problems in the intended application? Moreover, how could they be customized and how is that described to the potential user?

A corresponding question, but one that has commercial and legal ramifications is: How can we ensure that these components would be used properly?

Suppose we are building reusable components. Does using an object-oriented programming system (OOPS) guarantee that the components will be easier to reuse and customize? Can we build extensible components (e.g., customizable, as well) if we do not use an OOPS? To the latter question, the answer is most likely yes since programming languages such as Ada are object-oriented and provide some limited extensibility.

Reusability implies an adherence to standards. For Lego blocks each plug and each socket has a standard diameter such that plugs fit into sockets. Moreover, plugs and sockets are laid out on each block in a uniform, standard way.

With software, it's a different story. A number of standards that promote interoperability and interaction have been developed, including CORBA, SOM, and DCOM. Unfortunately, many of these standards are not interoperable among themselves. So a designer building for one standard will not produce components usable with another standard unless he creates a duplicate set oriented to the second standard.

Lack of a uniform interoperability and interaction standard is one of the great obstacles to truly reusable components.

11.2.2 Finding Components

Suppose we have a collection (or even several collections) of reusable components. How do we, as users, find the components that are right for us? The finding process is more than just locating an exact match. We want to find similar components, because if a component must be redeveloped for the target application, an example similar to the desired component can reduce the effort and eliminate many defects.

Suppose the components are described in a catalog just like Lego blocks, automotive parts, and integrated circuits. How do we determine that a particular component is a good fit to the requirements we have? One aspect of this is determining exactly what the component will do and, equally as important, what it will not do. A significant obstacle to reusable components is the lack of standardized descriptions that enable potential clients to match up their requirements with the capabilities of the component. So we need a standard for describing components that is susceptible to automation. While the model used to describe design patterns is a good start, it does not capture all the information we need, particularly about implementation.

Another obstacle is that components become less and less reusable the more specific they become. It becomes increasingly difficult to find an exact match for the user's detailed requirements. Solving the description problem is only part of the solution, because we still need to figure out how much detail about each component we provide for matching. If we can factor out the specificity, we can reduce the size of the search space considerably.

11.2.3 Using Components

Suppose we have found the components that we need to build our application. We have three tasks to perform. First, we must understand the components. Second, we must modify them if we need additional functionality. Third, we now have to assemble them into an application.

Understanding the component(s) is required whether or not we intend to modify them, but it is especially required if we do intend to modify them. The user needs a mental model of the component's computation to use it properly. This is a fundamental problem that limits our reuse of components. Lack of understanding and comprehension is due to many factors, but poor documentation and awkward code are two that come to mind.

Some components, of course, cannot be reused right out of the box. Some modification is required—most likely extensions and specializations of the component. The problem, of course, is how to write code such that components can be substituted for an alternative and changes can be made to the code when both new requirements arise and new components are available with minimal disturbance to the whole application.

Modification is a key process in software reusability. Modification varies with the programming language paradigm. This is largely a human skill, as there are almost no tools that provide any measure of automation of components.

The composition and assembly process is the most challenging one. On the one hand, we need to be able to specify the structure of the application and the generic components that comprise it. But we must also be able to specify the requirements for each concrete component that becomes part of the application. The problem is that component composition has both global and local effects, where the former are much harder to characterize and describe.

Do the components have an easy to use interface, like Lego blocks, that allows us to snap them together? If not, additional work is required to "glue" the components together, which may include writing additional software ("glue code") to perform data translation, data restructuring, or additional computations. This may require skilled IT professionals to develop applications, a scenario that components were originally intended to largely avoid.

11.2.4 Testing Components

Testing is a crucial software development activity that is used to determine if a software artifact has errors. Testing is used to assess the compliance of the implementation with the specifications. Two problems arise:

1. *Test Effectiveness:* What is the best selection of test data?
2. *Test Adequacy:* How do we know that sufficient testing has been performed?

Methodologies for testing software artifacts are addressed comprehensively in Beizer (1983), Hetzel (1988), and ANSI/IEEE (1989). Software testing requires both detailed knowledge of the specification and knowledge of the implementation. However, since we cannot observe the source code of a component, any testing that we do will be limited. So we must assess the testing of commercial off-the-shelf (COTS) components, which we cannot fully test ourselves.

At best, we can test the COTS component for domain compliance knowing the domain for which it was developed. Typically, we do not have access to the component's specification. We must use the component's interface description and accompanying documentation to guide any testing we want to do of the component.

Domain testing evaluates two properties of a component: its observable behavior and its controllable behavior. Observable behavior requires that every distinct input produce a distinct output. So by giving the component distinct inputs, we can determine the mapping of inputs to outputs. Controllable behavior requires that we can obtain a desired output value by specifying an input that yields the output. A key aspect is determining the sequences of inputs that yield particular outputs.

11.2.5 Harvesting Components for Reuse

While the concept of reuse is generally accepted as a good idea, few organizations have really attempted it on significant scale. In part, this is because few software projects have anything that is readily available to reuse.

The current investment in software may yield some components that can be reused in other applications. The challenge is to identify, extract, and package those components for reuse. Organizations need to create a pool of components to reuse. To do so, they must analyze existing software projects—one by one—to determine what might be reused.

Ideally, the reusable components should be managed through a catalog so that they are susceptible to searches as described in Section 11.2.2. The catalog is built by extracting metadata from individual components for entry into the catalog. You must define a classification scheme for the catalog that enables you to represent the characteristics of components to be represented there such that you can find components that will be useful to you.

The harvesting process for components has three steps:

1. Analyze existing applications and software projects to determine data structures and components that occur in a similar form across multiple systems.
2. Determine which of these recurring structures and components can be used to build future systems.
3. Reengineer and repackage existing components for reuse; enter their metadata in the catalog.

11.3 PROBLEMS IN USING COMPONENTS

Although we have been dealing with components for over ten years, we are still learning how to build component-based systems and frameworks. There are several problems that we'll discuss that affect using components. The principal problem is the glue: How do we wire components together? Software architectures, described in Part III, help to solve part of this problem.

11.3.1 Interoperability

As components are used to build more complex systems, it is no longer sufficient that components just be integratable. They must also be interoperable. Interoperability can be defined as the ability of two or more components to communicate and cooperate despite differences in the implementation language, the execution environment, or the abstraction model (Wegner 1996).

In order to solve the problem of describing interoperability, we need to know what information is required about interoperability. Two types of information are required: syntactic (static) interoperability and semantic (dynamic) interoperability.

Syntactic Interoperability Before components can interoperate, they must, of course, integrate. What do we need to know to integrate components? Primarily, we need to know the syntax of the interface to the component, for example, the syntax of the API. This syntax can be described using a formal method or a description language such as the CORBA IDL. Syntax enables structural integration, but not functional integration. So syntactic interoperability is the ability of two components to work together if properly connected; that is, data and messages passed between them are understood by each component.

Additional information is required to help potential users understand how to use the component and its provenance. This includes version information and dependencies of components. It specifies exactly the context in which the component was developed, including which versions of other components it requires, and which documentation and other information is associated with it. While this information is available during development, it is often lost once components are packaged and made available for deployment. This information allows the potential user to determine what types and versions of components work together.

Semantic Interoperability Interoperability also requires that we have semantic information about a component's functionality (Heiler 1995). This type of interoperability means that the behavior provided by a component should be in accordance with the behavior expected by its clients. One approach has been Bertrand Meyer's (1997) "design by contract."

No general established way of describing complete semantic information about components exists. Its implementation, the source code, is often the only existing formal description of a component's semantics. Software engineering is replete with semantic mismatches that lead to system failures such as the Ariane crash and the Mars Polar Lander (Huckle 2003).

Let us consider interoperability through an example using Figure 11-1.

Figure 11-1 depicts two processes that communicate. Process 1—consisting of components X1 and X2—communicates with process 2—consisting of components A1 through A4. The communication is effected from component A1 to component X2.

The two processes could be "owned" by different developers. Process 1 could represent a library that is used by process 2. A2 and A3 are other components used by A1, while A4 is used by A2 and A3. A2 and A3 may have been acquired commercially and brought along A4. If A2 and A3 were acquired from different sources, the developer needs to ensure that the versions of A4 are compatible and needs to choose one for inclusion in the system.

There are two levels of semantic interoperability that must be considered. The first is data interoperability. The two components can exchange data through some mechanism (such as RPC or messaging). This allows data to be shared by different objects and procedures independently of the programming languages or hardware platforms used. The type model is defined using the specifications based on the structures of the objects, and type compatibility is achieved by overcoming representation differences

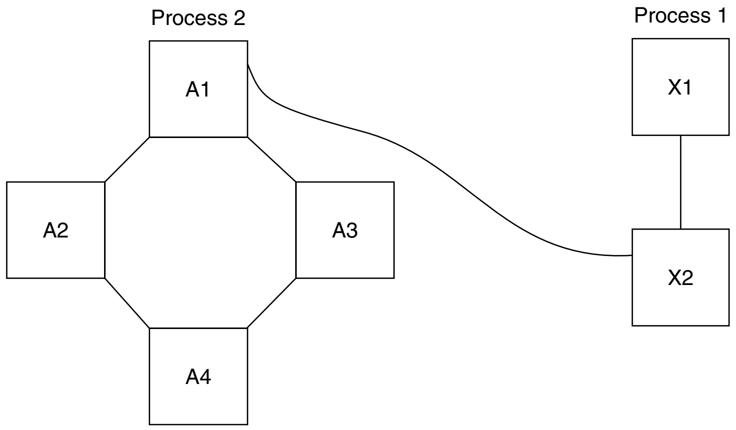

Figure 11-1. Component interoperability.

(such as byte ordering and floating point format) in how machines implement such objects, usually by translation into and out of a universal representation format. Most systems today provide application programming interfaces (APIs) to encapsulate such differences when various programs must interoperate in a networked environment, at least at the level of simple data types intrinsic to most popular programming languages (Howie et al. 1996).

The second level of interoperability is behavioral interoperability in which metadata about the data must be exchanged. This allows the participating components to understand how each interprets the data that is exchanged. For example, if a client or user agent were to search for information about a specific book from a set of distributed library resources, it would include an explicit indication that the search criteria is to be interpreted within the context of the library discipline. The search criteria could specify that the usage of a search term be restricted to a specific author or time period, the context restricted to within a copyright notice, and the authority restricted to a well-known discipline thesaurus. The client may also specify that the retrieved metadata be based on the library discipline. The requested search results may thus be restricted to retrieving the author's name, the time period of the book, the location of the original work, the location of reproductions, and a list of related books.

Two key questions arise:

1. How does one express interoperability information?
2. How does one publish this information so others can find it?

Both of these must be resolved before semantic interoperability can become independent of a particular implementation or technology.

One approach to this problem is the Z39.50 search and retrieval protocol, which has been designed to achieve semantic interoperability among heterogeneous data sources. Z39.50 has predefined a global set of search criteria and metadata, but it also offers a means for specialized disciplines to define their own set of search criteria and metadata. A number of disciplines have already established agreements, including Bibliographic Data, Government Information Resources, Scientific and Technical Data, Earth Sciences and Data, Digital Library Collections, and Museum Information (St. Pierre 1996).

Protocol Interoperability A third type of interoperability is called protocol interoperability (Yellin and Strom 1997), which addresses the relative order in which methods should be called in a component, the order in which it calls methods in other components, and the rules that govern component interactions. Two components are compatible if the restrictions imposed on the interaction of each component when they call each other are preserved; that is, their protocols match each role they share, and their communication is deadlock free.

Protocol interoperability impacts whether or not we can substitute one component for another. Suppose component A is to be replaced by component B. We must check two properties of the replacing component. First, all messages accepted by A (i.e., supported operations) are also accepted by B. Also, all of B's invoking messages, issued when implementing A's services (i.e., both the replies and the required operations), are a subset of the outgoing messages of A. This allows B to provide additional services beyond A's, but it must at least provide A's services. Second, we must test that the relative order among the incoming and outgoing messages of both components is consistent.

11.3.2 The Impact of Hidden Assumptions

Every developer of components makes some assumptions as he designs and implements the components. Such assumptions include decisions about garbage collection, persistence, concurrency, and sequencing of invocations to component services. A key problem is how to make these assumptions known to the users of components. Note that this does not mean revealing the internal implementation of the component, but providing to the user some indication of the performance characteristics of the component.

Every user of COTS components must deal with several uncertainties. First, there is uncertainty over the safety and security of the component. Second, you are unaware of the vendor's assumptions about how the component will be used. These assumptions may be in conflict with characteristics of your environment. These two concerns are further amplified by a concern over testing. Most COTS components are tested using the black-box method; white-box testing is not an option. Also, components provide little or no malleability for end-users to make improvements. Few components provide parameters that you can set that allow you to make precise improvements in the performance and/or behavior of the component. Finally, shrink-wrapped component licenses offer you only a "take it or leave it" option, which provides you with little recourse in the event of software failures.

11.3.3 Component Configuration Management

Users need to be able to find out about the components in order to find a suitable and consistent set of components for a given problem.

11.3.4 Dependability of COTS Components

Commercial off-the-shelf (COTS) software is software that is acquired from an independent, third party and is used on an "as is" basis. This definition encompasses a wide variety of items from utilities to DLLs to operating systems. Much COTS software comes shrink-wrapped, so that it can be installed right out of the box.

Using COTS components involves accepting and mediating a number of risks, including:

- Given a set of COTS components, fitting the application to them may be a significant challenge.
- Detailed specifications for the COTS components are often unavailable.
- The application developer may have questions about the components that cannot be answered, so the developer makes assumptions about what a component does.

- Some required functionality may be missing (e.g., it does 80% or 90% of what is needed).
- Some existing functionality may need to be masked (e.g., by wrapping).
- Unanticipated limitations and faults may arise in certain environments.
- You don't control the maintenance and evolution of the components.

A major problem we face when using a COTS component is how to ascertain its dependability. This has a number of ramifications: legal, ethical, and technical. From a legal standpoint, consider the following question: If an information system sold or delivered by you contains defective COTS components, will your clients absolve you of blame for system failures? Similar questions arise in the ethical arena. From a technical standpoint, the literature cites an average of between 0.5 and 2.0 defects per thousands of lines of code (KSLOC). As a vendor, can you live with this level of software defect in your product?

So what do you do? In most cases, the option of avoiding COTS by "rolling your own" is not acceptable. It takes too long, you miss urgent deadlines, and you still have many of the same testing and dependability issues. You could do an audit at the vendor site, but this is impractical because you can't afford to do it at every vendor's site that you might use. There will be some vendors whose products you have tested who you do not choose to use. They probably benefit more than you do in this case.

So what do you do? You can practice defensive programming by analyzing the ways a COTS component might fail in your system and surrounding it with wrappers to mitigate those failures. Reread the discussion on the Wrapper design pattern of Section 4.2 to see how this might be used in your environment.

11.4 PROBLEMS WITH GLUE CODE

Building applications from predefined software components requires some mechanism for connecting those components together in such a way as to enable them to collaborate and communicate. This mechanism is commonly known as *glue code*. As noted previously, the most difficult problem in using component technology is the glue. In this section we address some of the problems with glue code. The role of glue code is depicted in Figure 11-2.

In order for glue code to assist in implementing applications, it must be configured through parameterization to integrate two or more components.

11.4.1 Mapping Between Object Models

Object-oriented technology does not have a standard specification. Different vendors implement object-oriented programming in different ways—at the programming language or component level. For example, the Smalltalk model is different from the C++ model which is different from the CLOS model—yet all purport to be object-oriented technologies.

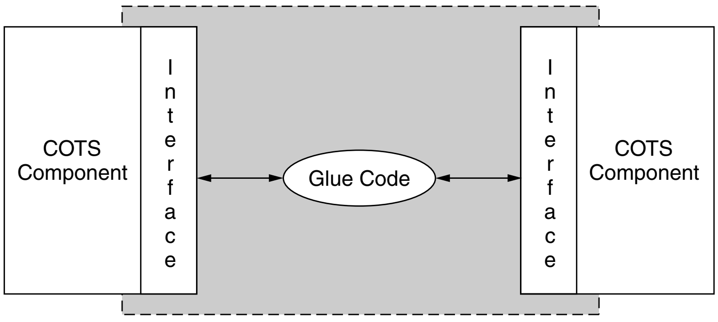

Figure 11-2. The role of glue code.

Glue code must translate, if necessary, between the different object models. CORBA provides such a feature, as does COM, although the latter uses a common calling sequence from different languages coupled with the COM interface mechanism. So CORBA supports (almost) general heterogeneity, while COM/DCOM only supports it within the Microsoft environment.

11.4.2 Adapting Between Incompatible Policies

Components may be developed with different types of policies related to security, interaction, storage, and so on. These policies must be made known explicitly in the component documentation in order for the programmer to understand and adhere to them, and understand the consequences of ignoring them. Glue code can assist in translating between policies, including masking out certain policies of the recipient component by providing default parameters.

11.4.3 Bridging Interaction Styles

Components also have different interaction styles. Ideally, one should be able to invoke component services in any order. However, realistically and practically, most components require that some methods be invoked in a certain order. Again, these requirements need to be made explicit in the component documentation. Glue code, acting as a wrapper, can accept method calls and reformat them into a different sequence of method calls in order to meet the requirements of the serving component.

11.4.4 Hooking into Legacy Components

Glue code also acts as a wrapper when interfacing with legacy components, perhaps written in other programming languages. It performs data translation, method or procedure argument arrangements, and sequencing of method calls for the legacy components.

11.4.5 Determining Good Glue

One of the problems we need to solve is determining what are good glue "abstractions." This is an open research problem in the sense that we need to develop good models for formally representing components in both client and server roles. Then we need to examine scenarios for different types of components and their features and properties.

11.5 EXERCISES

11.1. What is the best way to define a component in order to certify its properties? [Research Problem]

11.2. What component-related programming language features are needed to best support multiple, different implementations for the same abstract component, so as to enable plug-and-play?

11.3. What is the best way to define a component to allow client programmers to control the component's performance? [Research Problem]

11.4. Given a particular programming language, which language features should be aggressively used—and which should be avoided—when designing components to be composable, certifiable, efficient, reusable, and tunable? [Research Problem]

11.5. Compare and contrast C++'s constructor and destructor operations, with Ada 95's initialization and finalization operations in support of component instantiation.

11.6. Some authors have suggested that system integration productivity tends to decrease if the number of COTS components is increased. Give three reasons why you think this is so and justify your answer.