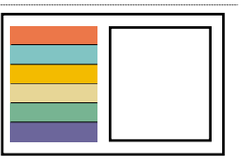
COMPONENT PRIMER



Laying the foundation.

JON HOPKINS

Software component technology has emerged as a key element in the development of complex software systems. As systems have grown larger, more complex and more interdependent, the characteristics of cohesion and coupling have become important drivers in the design and deployment of these systems. While the concept of software components has been well known virtually since the beginning of software, the practical aspects and challenges have been more fully evolved over time.

For the purposes of this overview, the following definition of component will be used: "A software component is a physical packaging of executable software with a well-defined and published interface." This abbreviated definition is formed from several other published definitions. For example, D'Souza and Wills define component as "A coherent package of software artifacts that can be independently developed and delivered as a unit and that can be composed, unchanged, with other components to build something larger" [1].

Similarly, Szyperski provides the following definition: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [3].

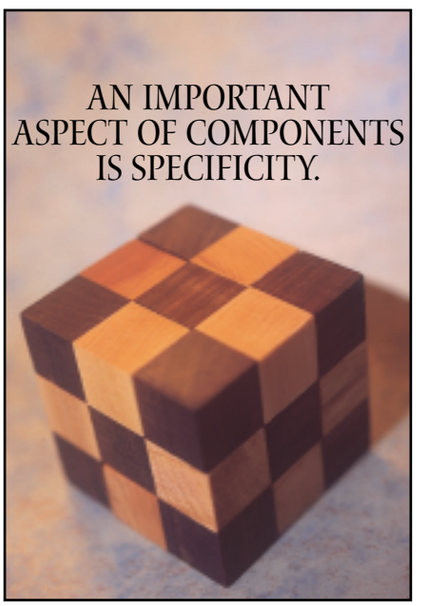
Other authors form similar definitions, with the focus on the interface and physical packaging. Previously, the definition of component was broad to include virtually any artifact of the software development process, such as documents, tests, source code, and parameter files. The definitions given here focus on the deployment aspect of components and their ability to be combined to form larger systems. In particular, the emphasis on well-defined interfaces, separate from their implementation, is critical to the success of components in loosely coupled systems.

Software developers have long held the belief that complex systems can be built from smaller components, bound together by software that creates the unique behavior and forms of the system. Ideally, a new system can be built using mostly predefined parts, with only a small

number of new components required. However, there are two engineering drivers in the development of a component-based system:

- **Reuse.** The ability to reuse existing components to create a more complex system.
- **Evolution.** By creating a system that is highly componentized, the system is easier to maintain. In a well-designed system, the changes will be localized, and the changes can be made to the system with little or no effect on the remaining components.

There are several caveats to the two forces mentioned. First, that there exist components to reuse. There must be a ready supply of well-built, applicable components that can be discovered, licensed, and easily used. There must also be a component model that can support the assembly and interaction of components—there must be a standard "backplane" in which the components can exist and communicate. Finally, there must be a process and architectures that support component-based development.



The notion of a component is a refinement of the concepts that have been taught as part of adopting object-oriented techniques—Meyer calls them a "natural extension" of the object model [2]. A component is a physical manifestation of an object that has a well-defined interface and a set of implementations for the interface. The component itself may or may not have been created using OO tools and languages. For example, it is perfectly possible to write a well-formed component in C, providing an interface and packaged implementation. However, practitioners often find the whole exercise of thinking about a system from a component perspective is most naturally modeled in an OO paradigm.

Object Models and Component Models

In order for component-based systems to work, it is necessary to have a component model on which to base the deployment and communication of the components. Two components can communicate only if they share a mechanism for finding each other and sending messages. In most operating systems, the common mechanism is some version of a common calling sequence for procedures, allowing procedures written in different languages to call each other. Like the object models found in various development tools and platforms, components need to have a reference model they can assume for purposes of interface definitions, message passing, and data transfer. There are currently several component models that have gained commercial support. In each case, the component models specify a means for components to publish their interface, send messages, and pass data.

The DCOM (Distributed Component Object Model) model from Microsoft has gone through several iterations and continues to evolve. While adapters to other component models exist, DCOM components are largely confined to the Windows platform (but are also implemented in several other environments by companies such as Hewlett-Packard).

The CORBA (Common Object Request Broker Architecture) model defined by the Object Management Group (OMG) is a language- and platform-independent specification, but has numerous language bindings and implementations on virtually all of the common platforms.

The Enterprise JavaBeans—or EJB specification—from Sun Microsystems provides a rich infrastructure for the execution of Java components.

While all three models share much in common, they are different enough that the selection of which component model to use in a project or enterprise is still strategic. In all three cases, there is an issue of platform dependencies. An alternative is to connect components through XML (Extensible Markup Language). Perhaps the most likely scenario will be that there will be a variety of component models suited to different platforms or environments, but they will be integrated through XML and transported over a variety of transport mechanisms. A component can then implement a single interface that accepts XML messages, and the component is then dependent only upon the format of the XML messages. This solution is suitable primarily for low-frequency, high semantic content exchanges, such as passing purchase order information, patient records, or complex requests.

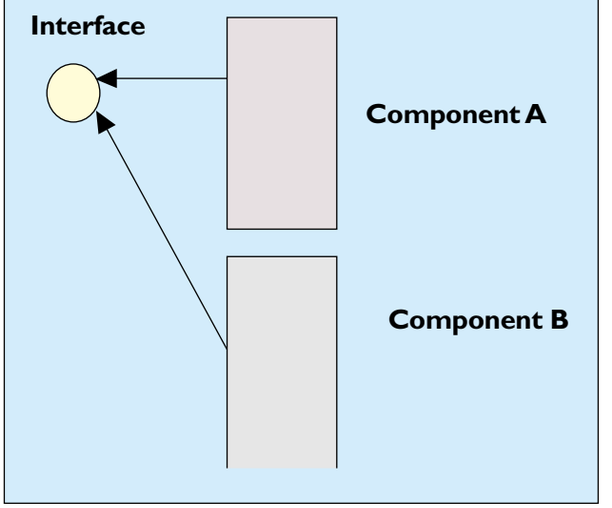
Modeling Components

The size and complexity of systems built using components necessitates some form of modeling. This is needed in order to comprehend the systems, communicate the design to others and to help manage the development process. The Unified Modeling Language (UML) has made explicit provisions for components in its metamodel. In particular, the component view allows architects to specify the component system that will realize a particular system or subsystem. The UML also supports the traceability between the class model, the component model, and the deployment model capturing the logical to the physical to the runtime. As multiple development groups build more systems from a variety of third-party components, or in collaboration, architects will spend more time considering the assembly of components, whether new or reused, and their interaction.

Ideally, most of the application developer's time is spent *integrating* components. This means the supplier of a component must provide accurate, detailed UML models of the component, particularly the interface, the state model, and the use cases the component supports (depending upon the granularity of the component). Without an accurate model, developers require source code to derive the equivalent information. Additionally, the models must specify extension points for the component to detail how the behavior of the component can be augmented, if at all.

One of the most important distinguishing factors in component-based development is the separation of the interface and the implementation. The interface itself can be registered and subsequently identified in searches. The interface is realized, or implemented, by one or more components, as shown in the figure appearing here.

Example interface implementation.



Components and Distributed Systems

Distributed systems, those systems that rely on the aggregate behavior of loosely coupled subsystems, are becoming more common. The Internet has introduced a transport mechanism that allows various disparate components to interact with each other to provide more complex, complete business behavior. System architects are now thinking at a higher level of abstraction, one that treats the individual components as the target platforms. The interactions between them form the dynamic behavior of the system. It appears the Net-based systems will depend upon a large set of services, manifested as interfaces, implemented in widely distributed components, supported by specialized vendors. An important enabling technology is the ability to interconnect the components in a platform-independent fashion. Rapidly evolving XML standards will provide a mechanism for components to share information. The messaging can take place over any number of different transports (IP, HTTP, RMI, and so forth).

Component Granularity

Component design decisions are driven by a variety of factors—foremost are several design constraints that help define the range of component granularity. Typically, intercomponent communication is fairly expensive in terms of time and platform resources. Thus, components are encouraged to be larger rather than smaller. However, larger components by nature have more complex interfaces and represent more opportunity to be affected by change. The larger the component, the less flexible is the structure of the system. So a balance is struck, depending upon the level of abstraction, likelihood of change, complexity of the component, and so forth. The principles of cohesion and coupling are the factors. Minimizing the coupling of the system tends to work against good cohesion.

Considerable examples of small components include many of the available GUI components (or "widgets") that are available from many small development shops. They are often implemented as ActiveX components or Java components because of the widespread nature of the component models, and because GUI components are typically executing on a client. Fewer components are designed for the CORBA environment because many of these are destined for commercial, proprietary systems.

Components can also be as large as whole appli-

cations, such as Excel or PowerPoint. These applications, often thought of as standalone programs, represent the set of services that make up the whole concept of spreadsheets or presentations, respectively. Existing applications that are to be integrated in a component architecture are usually "wrapped" in a component layer. The wrapper maps the existing interface, which can range from a well-organized API to raw file transfer, to an interface consistent with the enclosing architecture.

An important aspect of components is specificity. How specific is the component to a particular task? Clearly, the more closely a component matches the design, the less modification is required. Naturally, the number of components increases as they become more specific. To address this issue, component vendors must offer a family of solutions that capture the subtle variances from a basic design but that work in context of a larger framework. The differences between the components may be small, but the value to the designer is greatly enhanced. The cost to the original developer to produce the variant is dramatically smaller than the cost of developing the component from scratch by a consumer.

Markets for Third-Party Components

It is logical to conclude a market for components would develop, as the number of suppliers increase, and the number of consumers building systems increases. Components are potentially reusable assets, but it is not guaranteed that a component will be used simply because it exists. Myriad issues slow the growth of a third-party component market, including the following:

Platforms. A given component may often be usable on only one operating platform, limiting its use to the domain of that platform, or requiring multiple implementations for each target platform.

Architecture. Components are typically useful only within the context of a larger, unifying framework that provides structure and semantics. If a system is not component-based, or has no identifiable architecture, available components will be less valuable, and in some cases may be more awkward as the system attempts to conform to the interface of the component.

Specificity. A constant tension exists for any component that at once forces it to be general in design, but specific in function. As a result, many designers attempt to build a small number of general components that are highly parameterized. Instead, a selection of components that capture the subtle distinctions between various uses is required, arranged in a carefully designed taxonomy.

Versioning. Frequently referred to as "DLL Hell," incompatible versions of components necessarily exist, and may compete with themselves. Two clients may depend upon the same interface, but not upgrade simultaneously to the new version of the component. See [3] for a complete discussion on this.

Quality. The components originate with third-party sources and can only be tested on a component-level basis, not within the context of the complete system. This means that the coupling between components must be much more carefully designed. Component that rely on state information passed in to the component will be much more susceptible to errors of corruption, even if thorough checking of preconditions is performed.

Summary

Component-based software development represents an important stage in the maturation of the field of software engineering. It shifts the focus from new software development to the integration of existing components to perform new tasks. At the same time it addresses the issues of large-scale system development in the areas of coupling, distribution, and multiple platforms. As the component marketplace forms, it becomes one of so-called "increasing returns" whereby the components become more valuable as more become available, precisely because they start to realize the potential of the component-based ideal. C

References

1. D'Souza, D. and Wills, A.C. *Objects, Components and Frameworks with UML: The Catalysis Approach.* Addison Wesley, Reading, MA, 1999.
2. Meyer, B. The significance of components. In *Beyond Objects, SD Online* (Nov. 1999).
3. Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Longman Ltd, 1998.

Jon Hopkins (jhopkins@blueprinttech.com) is the chairman and chief technology officer at Blueprint Technologies, in McLean, VA.