4.4 What is a Component?

The key to understanding CBD is to gain a deeper appreciation of what is meant by a component, and how components form the basic building blocks of a solution. The definition of what it means to be a component is the basis for much of what can be achieved using CBD, and in particular provides the distinguishing characteristics between CBD and other reuse-oriented efforts of the past.

For CBD, a component is much more than a subroutine in a modular programming approach, an object or class in an object-oriented system, or a package in a system model. In CBD the notion of a component both subsumes and expands on those ideas. A component is used as the basis for design, implementation, and maintenance of component-based systems. For now we will assume a rather broad, general notion of a component, and define it as:

*An independently deliverable piece of functionality providing access to its services through interfaces.*

This definition, while informal, stresses a number of important aspects of a component. First, it defines a component as a deliverable unit. Hence, it has characteristics of an executable package of software. Second, it says a component provides some useful functionality that has been collected together to satisfy some need. It has been designed to offer that functionality based on some design criteria. Third, a component offers services through interfaces. To use the component requires making requests through those interfaces, not by accessing the internal implementation details of the component.

Of course, this definition is rather informal and provides little more than an intuitive understanding of components and their characteristics. It is in line with other definitions of a component [11, 12], and is sufficient to allow us to begin a more detailed investigation into components and their use. However, it is not sufficient for in-depth analysis of component approaches when comparing different design and implementation approaches. Consequently, a more formal definition of a component is provided later in this book.

To gain greater insight into components and component-based approaches, it is necessary to explore a number of topics in some detail. In particular, it is necessary to look at components and their use of object-oriented concepts, and view components from the perspective of distributed systems design. Based on such an understanding, the main component elements can then be highlighted.

Components and Objects[1]

[1] This discussion is based on John Daniel's excellent short paper on objects and component[13]

In discussions on components and component-based approaches there is much debate about components in relation to the concept of objects and object-oriented approaches. Examining the relationship between objects and components provides an excellent starting point for understanding component approaches [13, 14].

For over 30 years there have been attempts to improve the design of programming languages to create a closer, more natural connection between the business-oriented concepts in which problems are expressed, and the technology-oriented concepts in which solutions are described as a set of programming language statements. In the past decade these attempts have led to a set of principles for software structure and behavior that have come to be called object-oriented programming languages (OOPs).

There are many variations in OOPs. However, as illustrated in Figure 4.1, there are a number of concepts that have come to characterize an OOP [15].
The commonly identified principles of object orientation are:

- **Objects**: A software object is a way of representing software in an idea, a thing, or an event according to a chosen set of principles; these five principles are the following:

- **Encapsulation**: A software object provides a set of services and manipulates data within the object. The details of the internal operation and data structures are not revealed to clients of the object.

- **Identity**: Every object has a fixed, unique “tag” by which it can be accessed by other parts of the software. This tag, often called an object identifier, provides a way to uniquely distinguish that object from others with the same behavior.

- **Implementation**: An implementation defines how an object works. It defines the structure of data held by the object and holds the code of the operations. It is possible for an implementation to be shared by many objects.

- **Interface**: An interface is a declaration of the services made available by the object. It represents a contract between an object and any potential clients. The client code can rely only on what is defined in the interface. Many objects may provide the same interface, and each object can provide many interfaces.

- **Substitutability**: Because a client of the object relies on the interface and not the implementation, it is often possible to substitute other object implementations at runtime. This concept allows late or dynamic binding between objects, a powerful feature for many interactive systems.

For the past decade object-oriented principles have been applied to other fields, notably databases and design methods. More recently, they have been used as the basis for a number of advances in distributed computing as an approach to support the integration of the various pieces of a distributed application.

Based on this analysis, a component can be seen as a convenient way to package object implementations, and to make them available for assembly into a larger software system. As illustrated in Figure 4.2, a component is from this perspective a collection of one or more object implementations within the context of a component model. This component model defines a set of rules that must be followed by the component to make those object implementations accessible to others. Furthermore, it describes a set of standard services that can be assumed by components and assemblers of...
component-based systems (e.g., for naming of components and their operations, security of access to those operations, transaction management, and so on).

Organizations can define their own component models based on the needs of their customers and the tools they use to create and manipulate components (e.g., Sterling Software's proprietary CS/3.0 standard for components developed using the COOL:Gen product). Alternatively, a number of more widely used component model standards are now available, notably the Enterprise JavaBeans (EJB) standards from Sun Microsystems, and the COM+ standard from Microsoft.

Figure 4.2. The relationship between components and objects.

In summary, we see that in comparing components to objects, a component is distinguished by three main characteristics:

- A component acts as a unit of deployment based on a component model defining the rules for components conforming to that model.
- A component provides a packaging of one or more object implementations.
- A component is a unit of assembly for designing and constructing a system from a number of independently created pieces of functionality, each potentially created using an OOPL or some other technology.

Components and Distributed Systems

The 1980s saw the arrival of cheap, powerful hardware coupled with increasingly sophisticated and pervasive network technologies. The result was a move toward greater decentralization of computer infrastructures in most organizations. As those organizations built complex networks of distributed systems, there was a growing need for development languages and approaches that supported these kinds of infrastructures.

A wide range of approaches to application systems development emerged. One way to classify these approaches is based on the programming level that must be used to create distributed systems, and the abstract level of transparency that this supported. This is illustrated in Figure 4.3.

Initial approaches to building distributed systems were aimed at providing some level of hardware transparency for developers of distributed systems. This was based on technologies such as remote procedure calls (RPC) and use of message-oriented middleware (MOM). These allow interprocess communication across machines independent of the programming languages used at each end.

However, using RPCs still required developers to implement many services uniquely for each application. Other than some high-performance real-time systems, this expense is unnecessary. As a result, a technique was required to define the services made available at each end of the RPC to
improve the usability of these approaches. To allow this, the concepts of object-orientation were introduced as described above. By applying these concepts, application developers were able to treat remote processes as a set of software objects. The independent service providers are now implemented as components. They are structured to offer those services through interfaces, encapsulating the implementation details that lie behind them. These service providers have unique identifiers, and can be substituted for others supporting the same interfaces. Such distributed object technologies are in widespread use today. They include Microsoft’s Distributed Component Object Model (DCOM) and the Object Management Group’s Common Object Request Broker Architecture (CORBA).

Figure 4.3. Different levels of transparency for distributed systems.

This approach resulted in the concept of an Interface Definition Language (IDL). This provides a programming language a neutral way to describe the services at each end of a distributed interaction. It provides a large measure of platform independence to distributed systems developers. The services are typically constructed as components defined by their interfaces. This provides remote access to these capabilities without the need to understand many of the details of how those services are implemented. Many distributed computing technologies use this approach.

However, many implementations supporting an IDL were not transportable across multiple infrastructures from different vendors. Each middleware technology supported the IDL in its own way, often with unique value-added services. To provide a middleware transparency for applications, bridging technologies and protocols have been devised (e.g., the Internet InterOrb Protocol (IIOP)). These allow components greater independence of middleware on which they are implemented.

Of course, what most people want is to assemble solutions composed of multiple components — independently deliverable pieces of system functionality. These components interact to implement some set of business transactions. Ideally, developers would like to describe the services offered by components, and implement them independently of how those services will be combined later on. Then, when assembled into applications for deployment to a particular infrastructure, the services can be assigned appropriate characteristics in terms of transactional behavior, persistent data management, security, and so on. This level of service transparency requires the services to be implemented independently of the characteristics of the execution environment.

Application servers based on the COM+ and EJB specifications support this approach. These standards define the behavior a component implementor can rely upon from the “container” in which it executes. When being deployed to a container, the application assembler is able to describe the particular...
execution semantics required. These can change from one container to another without impact on the component’s internal logic, providing a great deal of flexibility for upgrade of component-based solutions.

Finally, it is possible in many domains to imagine common sets of services that would frequently be found in many applications within that domain. For example, in areas such as banking and financial management it is possible to construct a common list of services for managing accounts, transfer of funds between accounts, and so on. This is the goal of application transparency. Groups of experts in an industry domain have attempted to define common services in the form of a library, or an interacting framework of components and services. As a result, application designers and implementers in that domain can use a pre-populated set of components when designing new application behavior. This increases the productivity of developers, and improves the consistency of the applications being produced.

In summary, from the perspective of distributed systems, components provide a key abstraction for the design and development of flexible solutions. They enable designers to focus on business level concerns independently of the lower-level component implementation details. They represent the latest ideas of over two decades of distributed systems thinking.

**Elements of a Component**

As a result of these analyses, the elements of a component can now be discussed. Building on object-oriented concepts and distributed systems thinking, components and component-based approaches share a number of characteristics familiar to today’s systems designers and engineers. However, a number of additional characteristics must also be highlighted. These are shown in Figure 4.4.

**Figure 4.4. What is a component?**

As illustrated in Figure 4.4, there are five major elements of a component:

- **A specification.** Building on the interface concept, a component requires an abstract
description of the services it offers to act as the contract between clients and suppliers of the services. The component specification typically defines more than simply a list of available operations. It describes the expected behavior of the component for specific situations, constrains the allowable states of the component, and guides the clients in appropriate interactions with the component. In some cases these descriptions may be in some formal notation. Most often they are informally defined.

- **One or more implementations.** The component must be supported by one or more implementations. These must conform to the specification. However, the specification will allow a number of degrees of freedom on the internal operation of the component. In these cases the implementer may choose any implementation approach deemed to be suitable. The only constraint is on meeting the behavior defined in the specification. In many cases this flexibility includes the choice of programming language used to develop the component implementation. In fact, this behavior may simply be some existing system or packaged application wrapped in such a way that its behavior conforms to the specification defined within the context of the constraining component standard.

- **A constraining component standard.** Software components exist within a defined environment, or component model. A component model is a set of services that support the software, plus a set of rules that must be obeyed by the component in order for it to take advantage of the services. Established component models include Microsoft’s COM+, Sun’s JavaBeans and Enterprise Java Beans (EJB), and the OMG’s emerging CORBA component standard. Each of these component models address issues such as how a component makes its services available to others, how components are named, and how new components and their services are discovered at runtime. Additionally, those component models concerned with enterprise-scale systems provide additional capabilities such as standard approaches to transaction management, persistence, and security.

- **A packaging approach.** Components can be grouped in different ways to provide a replaceable set of services. This grouping is called a package. Typically, it is these packages that are bought and sold when acquiring components from third-party sources. They represent units of functionality that must be installed on a system. To make this package usable, some sort of registration of the package within the component model is expected. In a Microsoft environment, for example, this is through a special catalog of installed components called the registry.

- **A deployment approach.** Once the packaged components are installed in an operational environment, they will be deployed. This occurs by creating an executable instance of a component and allowing interactions with it to occur. Note that many instances of the component can be deployed. Each one is unique and executes within its own process. For example, it is possible to have two unique instances of an executing component on the same machine handling different kinds of user requests.