Large scale CBD

Monday, January 13, 2020 3:34 PM

BOOK

Large-Scale, Component-Based Development

By Alan W. Brown

Write the first review

Prentice Hall May 2000

3.5. Components and Component Models

Current trends are moving more enterprise-scale solutions toward the use of the Internet as a cost-effective infrastructure for distribution of these applications. With this goal in mind, development techniques for enterprise-scale solutions in the Internet age must support:

- Services that are designed and implemented in pieces by teams of developers;
- Deployment of these pieces across a range of machines;
- Integration of functionality derived from many sources, including purchased packages and previously developed systems;
- Targeting n-tier architectures involving web servers and applications servers.

Supporting these needs is the goal of components and component-based approaches to providing software solutions. Component-based approaches address two main concerns. The first is how to design and assemble solutions as a collection of interacting pieces. To do this requires techniques for describing the overall system architecture, for carving that into appropriate constituent pieces, and allowing each piece to be designed and provisioned in a way that facilitates its assembly with the other pieces.

Second, a standard set of services is required to handle the problem of accessing pieces of a distributed application when the location of each piece cannot be determined until runtime. To allow this, there must be standard services for naming, locating, and accessing these pieces. Additionally, to assemble the pieces into a complete application requires a standard way to identify which operations a given module will support, and a robust way to pass messages between the pieces to invoke those operations.

Thus, enterprise-scale solutions in the Internet age must be component-based. Consequently, products and services supporting web-based solutions must be based on a component standard.

As a result, when organizations start to provision distributed web-based solutions, they tend to move to components and to an n-tiered model that places business logic in components that reside on the middle tiers. Over the past few years this approach has been widely adopted by end-user organizations. It is supported by key standards from Microsoft through its COM+ initiative [13], Sun through its Enterprise JavaBeans (EJB) specification [14], and the Object Management Group (OMG) with its efforts to standardize a language and technology-neutral open server component model [15].

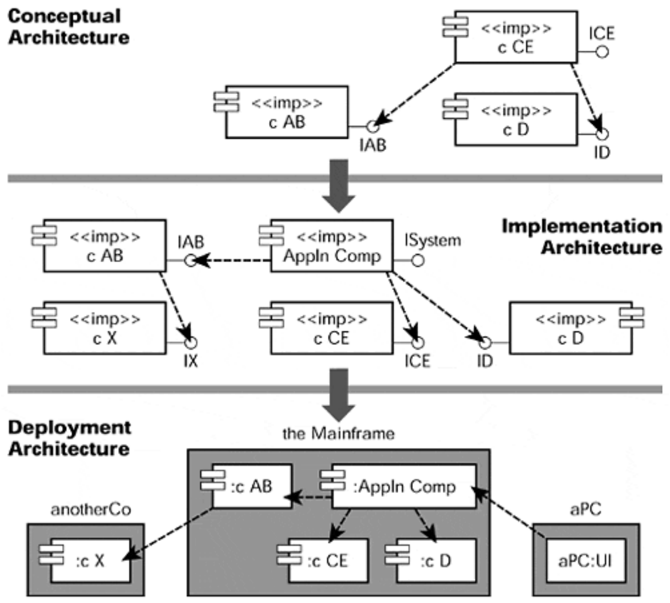
However, a component-based approach is more than simply deploying components to the middle tiers of an n-tier architecture. The application server is a platform for hosting distributed systems and integrating different kinds of assets across the corporation. To make use of this, distributed systems platforms require:

- A way to see the conceptual and physical architecture of the solution as a set of interacting pieces;
- A design approach supporting distributed systems provisioning that focuses on how the system should be partitioned for the greatest flexibility;
- A direct mapping from the system design into the underlying implementation technologies, supported by the application server.

The way to achieve this is to design with components to create solutions as sets of logical pieces, interacting via their defined interfaces, and to provision with components to create physical packages of system behavior for deployment to an infrastructure supporting a well-defined component model.

Designing with Components

Component approaches concentrate design efforts on defining interfaces to pieces of a system, and describing an application as the collaborations that occur among those interfaces. The interface is the focal point for all analysis and design activities. Implementers of a component can design and build the component in any appropriate technology as long as it supports the operations of the interface and is compatible with the component execution environment. Similarly, users

of another component can define their use of that component by reference only to its interfaces. This provides a measure of independence between component developers, and improves the flexibility of the system as the components are upgraded and replaced.
As a result, the components can be provisioned using a variety of technologies, including wrapping some existing code or data, using the application programming interface (API) of a purchased package, or writing new purpose-built logic into an application development tool. As long as the behavior defined by the interface is supported, a client for that component does not need to be aware of the way in which that behavior has been implemented internally.
Solutions are designed as collections of collaborating components. However, there are different abstract levels on which this component design may be considered. At a conceptual level, a system may be considered to be a number of logical pieces supporting each major functional area of the system. However, at an implementation level, each logical component of the system may be supported by one or more physical components. These may have been developed in a variety of technologies. Finally, these physical components will be deployed to a set of computers in a distributed network. Many variations of their deployment are possible based on system requirements such as performance and availability.
In Figure 3.6 we illustrate the kinds of component architecture diagrams important in understanding how a distributed system is designed and deployed. The figure shows that different kinds of component architecture are needed, including conceptual, implementation, and deployment architectures. This concept of component architecture will be examined in detail in later chapters.
Figure 3.6. Example of component architecture diagrams.


Implementing for Components
As highlighted earlier in this chapter, many organizations building distributed web-based applications develop solutions targeting an N-Tier architecture consisting of clients, web server, and potentially multiple levels of application servers and back-office systems. Consequently, the applications targeting these architectures are developed in pieces, and must be deployed and managed as a set of interacting services executing on these different tiers. Components and component technologies provide the enabling approaches for developing these applications in a robust, repeatable way based on industry standards.
It has been found that a key element for bringing together these pieces is the application server. Much of the logic necessary to tie together the various pieces is embedded in the components executing on the application server. As a result, the application server is essentially an integration infrastructure in which components can be executed. Because of the importance of components to web-based applications, many of the available application server technologies have become specialized toward supporting one of the many component standards that have been developed.
The component model defines a set of standards for designing, assembling, and deploying applications from pieces. It describes how each component must make its services available to others, how to connect one component to another, which common utility services can be assumed to be provided by the infrastructure supporting the component model, how new components announce their availability to others, and so on. In short, by supporting a particular component model the application server is providing a programming approach for developers to use that application server.
Three particular component models currently dominate the market: COM+, CORBA, and Java/EJB. These are summarized in Table 3.1, adapted from [4].

Standard	Maturity	Scalability	Pace of Innovation	ISV Mind Share	IT Mind Share	Summary
COM+	MTS is several years old	Departmental, enterprise in some cases	Fast, based on product needs	Leader	Emerging leader	Increasing popularity based on Windows domination
CORBA	Mature specs; immature service implementations	Enterprise in most cases	Slower, based on consensus	Laggard	Fading leader	Looking to tie in with Java and EJB direction
Java/EJB	Immature; lots of current activity	Departmental now, but aimed at the enterprise	Fast, based on rapid Java adoption and catch-up to Microsoft	Emerging leader	Emerging leader	Client-side Java now, giving way to server-side Java

Table 3.1. A Comparison of COM+, CORBA, and EJB from an EAI Perspective

Microsoft's dominance on the desktop is based on its Windows operating systems: Windows 95, Windows 98, and Windows NT. With its latest releases, Microsoft is positioning Windows 2000 as an application server platform based on its support for COM+, a combination of the COM object model, the Microsoft Transaction Server (MTS) for transaction management, and a programming model placed around these supported by events, asynchronous messaging, dynamic load balancing, and life-cycle management services. It offers a comprehensive approach to enterprise-scale application development targeting the Microsoft platforms.

The Object Management Group (OMG) has been developing standards for distributed systems for a number of years. It has been most successful with its Common Object Request Broker (CORBA) standard, and a number of implementations of that standard. This has been available for a number of years, supported by a growing number of services and tools. It is now about to release a component model for CORBA, heavily influenced by the Java/EJB standard. Products based on the OMG standards have been successful in their focus on supporting integration of heterogeneous platforms to create enterprise-scale systems.

Sun has had a great deal of success with its Java programming language, and the many services defined for building distributed systems for the Internet using Java. As part of the standardization of Java, Sun recently released the first version of its server-side standard for Java, Enterprise Java Beans (EJB). More than 50 companies have already announced that they will support the EJB specification by offering application servers supporting the EJB standard. This provides one of the final pieces for a complete end-to-end story for the use of Java as an enterprise application development language. With its promise of application portability across many application servers, many organizations are investigating EJB-based solutions as key technology for developing future web-based enterprise-scale systems.

Chapter 4. Component-Based Development Fundamentals

Components and component-based approaches are the driving force for the e-business revolution. They are the way enterprise-scale solutions for the Internet age will be developed. In a recent report [1], Gartner Group predicted that by 2003 up to 70% of all new software-intensive solutions will be constructed with "building blocks" such as prebuilt components and templates. Similarly, in its 1999 survey of worldwide markets and trends [2], International Data Corporation (IDC) estimated that by 2003 the worldwide market for component construction and assembly tools will be in excess of $8 billion, with an additional $2 billion spent on acquiring components. As a result, it is clear that over the next few years enterprise-scale solutions in the Internet age will undergo fundamental changes in both capability and focus.

Consequently, it is essential to understand the concepts and ideas that underlie the design and implementation of component-based solutions. Furthermore, this insight must be coupled with knowledge of many of the practical issues to be addressed in making a component approach succeed in the context of today's tool and infrastructure capabilities. This part of the book addresses the concepts critical to components and component approaches. The next part of the book deals with practical issues of the application of these concepts, and the technologies required to make it successful.

4.1. Introduction

Complex situations in any walk of life are typically addressed by applying a number of **key concepts. These are represented in approaches such as abstraction, decomposition, iteration, and refinement.** They are the **intellectual tools** on which we rely to deal with difficult problems in a controlled, manageable way.

Critical among them is the technique of **decomposition**—dividing a larger problem into smaller, manageable units, each of which can then be tackled separately. This technique is at the heart of a number of approaches to **software engineering**. The **approaches** may be called **structured design, modular programming, or object-orientation**, and the **units** they produce called **modules**, **packages**, or **components**. However, in every case the main principle involved is to consider a **larger system** to be **composed** from well-defined, **reusable units** of functionality, introduce ways of managing this **decomposition** of a **system** into **pieces**, and enable its subsequent reconstruction into a **cohesive system**.

While the concepts are well understood, most organizations struggle to apply these concepts to the provisioning of enterprise-scale solutions in the Internet age. However, driven by the challenges faced by software engineers today, many organizations are beginning to reassess their approach to the design,

implementation, and evolution of software. This stems from growing pressure to assemble solutions quickly from existing systems, make greater use of third-party solutions, and develop reusable services for greater flexibility.
Consequently, renewed attention is being given to reuse-oriented, component-based approaches. This in turn is leading to new component-based strategies being defined, supported by appropriate tools and techniques. The collective name for these new approaches is Component-Based Development (CBD) or Component-Based Software Engineering (CBSE). For developers and users of software-intensive systems, CBD is viewed as a way to reduce development costs, improve productivity, and provide controlled system upgrade in the face of rapid technology evolution.
In this chapter we examine the renewed interest in reuse-oriented approaches, and define the goals and objectives for any application development approach able to meet the needs of today's software development organizations. Armed with this knowledge, we examine the major principles of CBD, and the characteristics that make CBD so essential to software developers and tool producers alike. We then consider the major elements of any CBD approach, and highlight the main concepts behind them.

4.2. The Goals of Component Approaches

Recently there has been renewed interest in the notion of software development through the planned integration of preexisting pieces of software. This is most often called component-based development (CBD), component-based software engineering (CBSE), or simply componentware, and the pieces called components.
There are many ongoing debates and disagreements concerning what exactly are and are not components [3]. Some authors like to emphasize components as conceptually coherent packages of useful behavior. Others concentrate on components as physical, deployable units of software that execute within some well-defined environment [4]. Regardless of these differences, the basic approach of CBD is to build systems from well-defined, independently produced pieces. However, the interesting aspects of CBD concern how this approach is realized to allow components to be developed as appropriate cohesive units of functionality, and to facilitate the design and assembly of systems from a mix of newly and previously developed components.
The goal of building systems from well-defined pieces is nothing new. The interest in CBD is based on a long history of work in modular systems, structured design, and most recently in object-oriented systems [5]–[9]. These were aimed at encouraging large systems to be developed and maintained more easily using a "divide and conquer" approach. CBD extends these ideas, emphasizing the design of solutions in terms of pieces of functionality provisioned as components, accessible to others only through well-defined **interfaces**, outsourcing of the implementation of many pieces of the application solution, and focusing on controlled assembly of components using interface-based design techniques.
Most importantly, these concepts have been supported by a range of products implementing open standards that offer an infrastructure of services for the creation, assembly, and execution of components. Consequently, the application development process has been **reengineered** such that software construction is achieved largely through a component selection, evaluation, and assembly process. The components are acquired from a diverse set of sources, and used together with locally developed software to construct a complete application [10].

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]

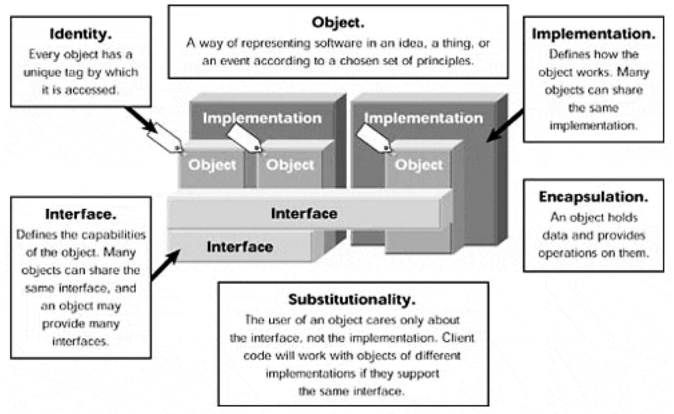
[1] This discussion is based on John Daniel's excellent short paper on objects and component[13]

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 (OOPLs).

There are many variations in OOPLs. However, as illustrated in Figure 4.1, there are a number of concepts that have come to characterize an OOPL [15].

Figure 4.1. Basic concepts of object-oriented approaches.


Identity.
Every object has a unique tag by which it is accessed.
Object.
A way of representing software in an idea, a thing, or an event according to a chosen set of principles.
Implementation.
Defines how the object works. Many objects can share the same implementation.
Implementation
Implementation
Object
Object
Object
Interface
Interface
Interface.
Defines the capabilities of the object. Many objects can share the same interface, and an object may provide many interfaces.
Encapsulation.
An object holds data and provides operations on them.
Substitutionality.
The user of an object cares only about the interface, not the implementation. Client code will work with objects of different implementations if they support the same interface.

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.

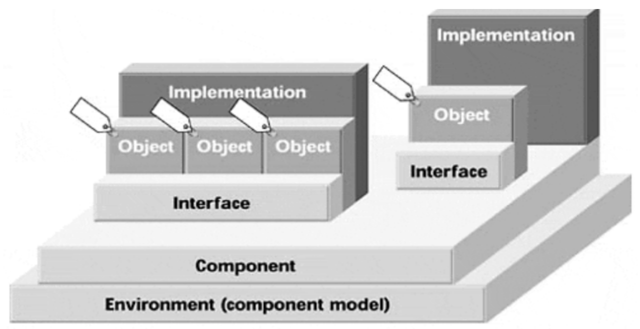
For the past decade object-oriented principles have been applied to other fields, notably databases and design methods. More recently, they have been used as the basis for a number of advances in distributed computing as an approach to support the integration of the various pieces of a distributed application.

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.

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

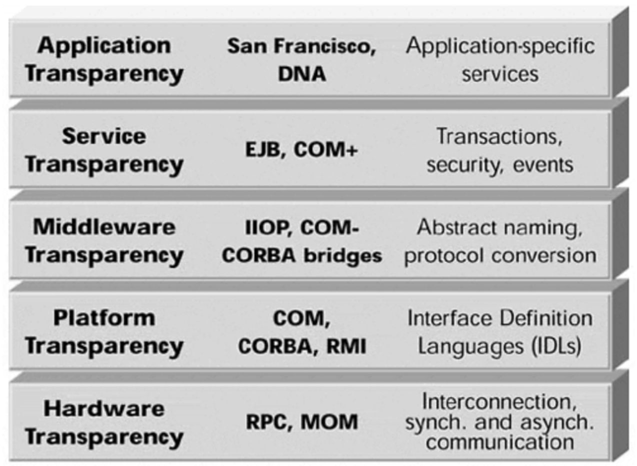
The 1980s saw the arrival of cheap, powerful hardware coupled with increasingly sophisticated and pervasive network technologies. The result was a move toward greater decentralization of computer infrastructures in most organizations. As those organizations built complex networks of distributed systems, 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.

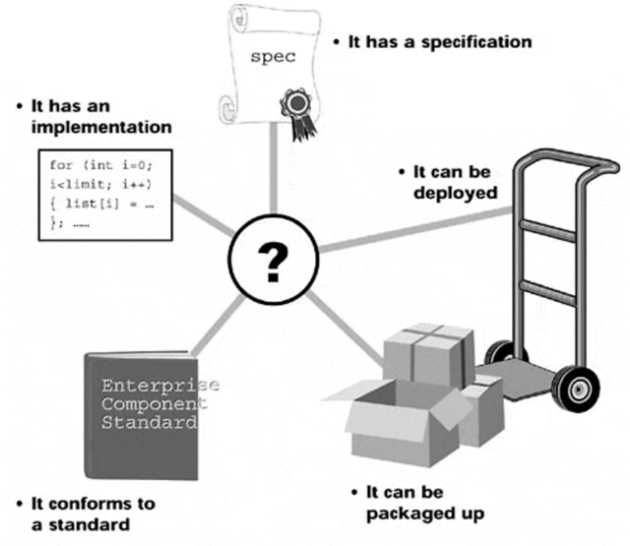
However, many implementations supporting an IDL were not transportable across multiple infrastructures from different vendors. Each middleware technology supported the IDL in its own way, often with unique value-added services. To provide a middleware transparency for applications, bridging technologies 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 Java Beans 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.

4.5. How are Applications Assembled Using CBD?

Having described the basic component concepts, we now consider how these concepts are embodied in supporting a component-oriented approach to application assembly. In particular, we identify the key elements on which any CBD approach is based.

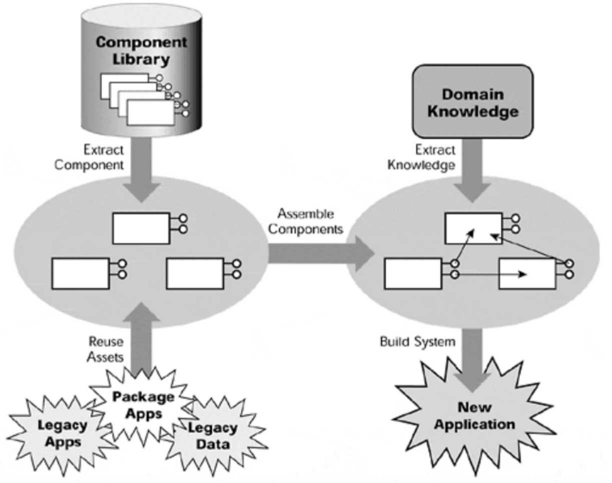
While much of the technology infrastructure for component-oriented approaches is in place, of equal importance to its success are the methods for developing component-oriented applications. Faced with the wealth of technology, software developers must be able to answer the key question of how to effectively design solutions targeting that technology. The software industry is only just beginning to offer guidance in this regard.

Fortunately, the latest wave of software development methods are beginning to rise to the challenge of supporting the development of distributed, web-based systems involving the reuse of legacy systems and packaged applications. A number of organizations have begun to publicize their methods and best practices for developing enterprise-scale systems from components. The three most prominent of these are:

- **Rational's Unified Process.** This is a broad process framework for software development covering the complete software life cycle. In architecting the solution, a component-based approach is encouraged, heavily influenced by the constraints of the Unified Modeling Language (UML) notation [16].
- **The Select Perspective Method.** A general component design approach is supported, targeted at the Select Component Manager. General component design principles target UML as the component design notation [17].
- **Sterling Software's Enterprise-CBD Approach.** Influenced by the Catalysis approach to component development, Sterling Software's approach encourages a strong separation of component specification from implementation using an extended form of UML [18]. This allows technology-neutral specifications to be developed and then refined into implementations in a number of different implementation technologies.

These approaches differ in many details. However, the abstract method that they encourage for component-based design is fundamentally the same. As illustrated in Figure 4.5, three key elements provide the focus of these methods: a diverse set of component stored in a component library, an interface-focused design approach, and application assembly based on a component architecture.

Figure 4.5. Elements of a component-oriented software process



Sources of Components

A component has been informally defined as an independently deliverable piece of functionality providing access to its services through interfaces. This definition is important as much for what it doesn't say as for what it does say; it does not place any requirements on how the components are implemented. Hence, valid components could include packaged applications, wrapped legacy code and data, previously developed components from an earlier project, and components developed specifically to meet current business needs.

The diverse origin for components has led to a wealth of techniques for obtaining components from third parties, developing them in a variety of technologies, and extracting them from existing assets. In particular, many CBD approaches focus on wrapping techniques to allow access to purchased packages (e.g., Enterprise Resource Planning (ERP) systems) and to mine existing legacy systems to extract useful functionality that can be wrapped as a component for future use. This approach leads to a view of CBD as the basis for enterprise application integration (EAI). Many vendors providing technologies for combining existing systems take a component view of the applications being assembled, and introduce specific integration technologies to tie the systems together. Often this is focused on the data flows in and out of the packages, and consists of transformers between the packages offering a broker-based approach.

The sources of the components may be diverse and widespread, yet assembling solutions from these components is possible due to the following:

- Use of a component model as a standard to which all components can conform regardless of their heritage;
- A component management approach, supported by appropriate tools, for the storage, indexing, searching, and retrieval of components as required;
- A design approach that allows solutions to be architected from components by considering their abstract functionality, and ignoring peculiarities of their implementation for a later date.

This final aspect, a design approach targeting appropriate component architectures, is a key element to the success of component approaches in general. In particular, it is the motivation for interface-based design approaches.

Interface-Focused Design

Interfaces are the mechanism by which components describe what they do, and provide access to their services. The interface description captures everything on which the potential client of that component can rely since the implementation is completely hidden. As a consequence, the expressiveness and completeness with which the interfaces are described is a primary consideration in

any component-based approach to software.
Furthermore, during the early stages of analysis and design, the focus is on understanding the roles and responsibilities within the domain of interest as a basis for the interfaces of the implemented system. This gives rise to a new form of design method: interface-based design. Focusing on interfaces as the key design abstraction leads to much more flexible designs [19, 20]. Designers are encouraged to consider system behavior more abstractly, define independent suppliers of services, describe collaborations among services to enact scenarios, and reuse common design patterns in addressing familiar situations. This results in more natural designs for systems with a greater independence from implementation choices.
Such thinking is an essential part of new application provisioning approaches in the Internet age. To target rapidly evolving distributed computing technologies requires an approach to design that can evolve as the technology changes. Familiar, rigid approaches to design are unsuitable. Interface-based design approaches represent a significant step forward in reducing the cost of maintenance of future systems.[2]

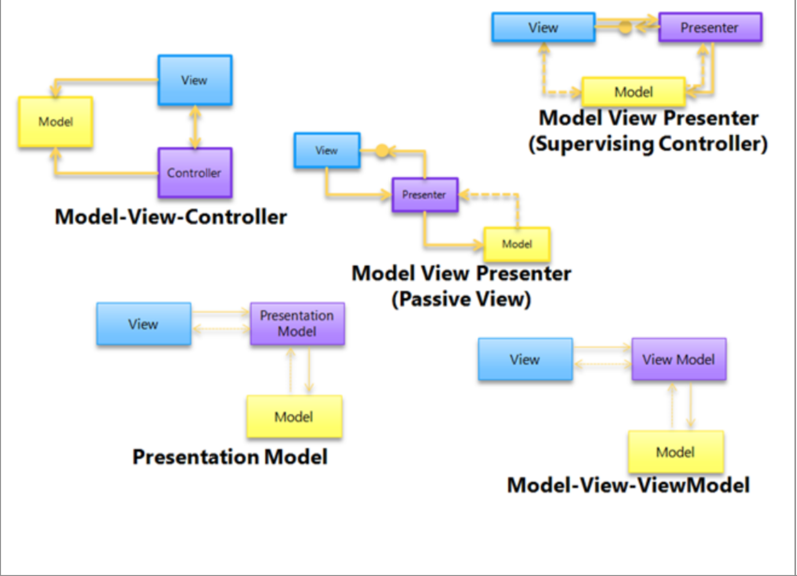
[2] Detailed examples of interface-based design approaches are provided later in this book.

Applications and Component Architecture

Interfaces and interface-based design provide the techniques necessary for a component assembly view of software-intensive solutions. Component assembly concerns how an application is designed and built from components. In a component-based world, an application consists of a set of components working together to meet the broader business needs. How this is achieved is often referred to as the component architecture—the components and their interactions. These interactions result in a set of dependencies among components that form an essential part of a software solution. Describing, analyzing, and visualizing these dependencies become critical tasks in the development of a component-based application.
Typically, a candidate component architecture is proposed relatively early in a project. This consists of a number of known pieces of existing systems to be reused, familiar patterns of system interactions, and constraints based on imposed project and organizational standards. Consequently, at least two levels of component architecture become important.
The first is a logical component architecture. This describes the abstract design of the system in terms of the major packages of functionality it will offer, descriptions of each collection of services in terms of its interfaces, and how those packages interact to meet common user scenarios. This is often called the component specification architecture. It represents a blueprint of the design of the system. Analysis of this architecture is essential to ensure that the system offers appropriate functionality, and can easily be modified as the functional requirements for the system evolve.
The second is a physical component architecture. This describes the physical design of the system in terms of selected technical infrastructure products, distributed hardware and its topology, and the network and communication protocols that tie them together. This forms part of the component implementation architecture. This architecture is used to understand many of the system's non-functional attributes such as performance, throughput, and availability of services.

6.4. Component Infrastructure Technology

To support a component-based approach, it is common to use some form of component infrastructure (sometimes also called "component-oriented middleware") to handle all of the complex details of component coordination [8]. Essentially, the component infrastructure provides a common set of component management services made available to all components interested in using that infrastructure. The component infrastructure imposes constraints on the design and implementation of the components. However, in return for abiding by these constraints, the component developer and application assembler is relieved from the burden of developing many complex services within their application.
To understand component infrastructures, it is necessary to understand the kinds of services the infrastructure can make available, and the different competing infrastructure implementations currently available.



The main component infrastructure standard provided by OMG is the Common Object Request Broker Architecture (CORBA) [10]. This defines the basic distribution architecture for component-oriented applications. There are three major aspects of CORBA:

- The OMG's Interface Definition Language (IDL), which describes how business functionality is packaged for external access through interfaces.
- The CORBA component model describing how components can make requests of each other's services.
- The Internet Inter-ORB Protocol (IIOP), which allows different CORBA implementations to interoperate.

Together with the CORBA standard, a set of additional capabilities is defined in the CORBA-Services standards [11]. A wide range of services has been defined, or is currently under investigation. However, the following services are those that are most often found in currently available implementations:

- Life cycle services, which control the creation and release of component instances.
- Naming services, which allow identification and sharing of component instances.
- Security services, which provide privacy of connection between a client and provider of services.
- Transaction services, which allow a user to control the start and completion of distributed transactions across components.

1. CORBA
2. Spring
3. .NET
4. Ruby and Rails
5. Angular
6. Node JS
7. JSF
8. Structs

Sun's Java-Based Distributed Component Environment

One of the most astonishing successes of the past few years has been the rapid adoption of Java as the language for developing client-side applications for the web [16, 17]. However, the impact of Java is likely to be much more than a programming language for animating web pages. Java is in a very advantageous position to become the backbone of a set of technologies for developing component-based, distributed systems. Part of this is a result of a number of properties of Java as a language for writing programs:

- Java was designed specifically to build network-based applications. The language includes support for distributed, multithreaded control of applications.
- Java's runtime environment allows pieces of Java applications to be changed while a Java-based application is executing. This supports various kinds of incremental evolution of applications.
- Java is an easier language to learn and use for component-based applications than its predecessors including C++. Many of the more complex aspects of memory management have been simplified in Java.
- Java includes constructs within the language supporting key component-based concepts such as separating component specification and implementation via the interface and class constructs.

However, Java is much more than a programming language. There are a number of Java technologies supporting the development of component-based, distributed systems. This is what allows us to consider Java as a component infrastructure technology [18].

More specifically, there are a number of Java technologies providing packaging and distribution services. These include [19]:

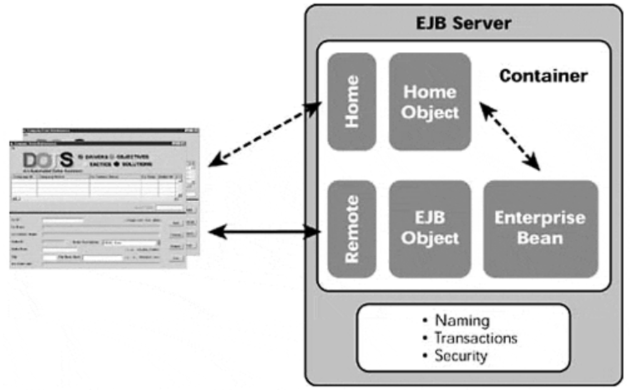
- JavaBeans, which is the client-side component model for Java. It is a set of standards for packaging Java-implemented services as components. By following this standard, tools can be built to inspect and control various properties of the component.
- Remote Method Invocation (RMI), which allows Java classes on one machine to access the services of classes on another machine.
- Java Naming and Directory Interface (JNDI), which manages the unique identification of Java classes in a distributed environment.

An additional set of technologies support the remaining component infrastructure services. These are necessary to allow Java to be used for the development of enterprise-scale distributed systems. These technologies are defined within the Enterprise JavaBeans standard. Enterprise JavaBeans (EJB) is a standard for server-side portability of Java applications [20]. It provides the definition of a minimum set of services that must be available on any server conforming to the specification. The services include: process and thread dispatching, and scheduling; resource management; naming and directory services; network transport services; security services; and transaction management services.

The goal of the Enterprise JavaBeans specification is to define a standard model for a Java application server that supports complete portability. Any vendor can use the model to implement support for Enterprise JavaBeans components. Systems such as transaction monitors, CORBA runtime systems, COM runtime systems, database systems, Web server systems, or other server-based runtime systems can be adapted to support portable Enterprise JavaBeans components. By early in 2000 all leading platform middleware vendors (except Microsoft) will have delivered (or claimed) support for the EJB standard. It is considered the primary alternative to Microsoft's COM+ model.

As illustrated in Figure 6.5, the EJB specification describes a number of key aspects of a system. In particular, a user develops an Enterprise JavaBean implementing the business logic required. The user also defines a home interface for the bean defining how the EJB objects (i.e., instances of the EJB) are created and destroyed, and a remote interface for clients to access the bean's behavior. An EJB executes within an EJB container. Many EJB containers can operate within a given EJB server. The EJB server provides many of the basic services such as naming, transaction management, and security.

Figure 6.5. Elements of an EJB server.



The EJB specification was made available early in 1999. However, by the end of 1999, over 25 vendors already offered EJB-compliant containers. These implement the component infrastructure services that any application developer can rely on when designing a component-based application in Java. The strong support for EJB from both Sun and IBM provides significant impetus to the case for EJB as an important player in the future of component infrastructure services.

Toward the end of 1999, Sun Microsystems announced a new initiative aimed at bringing together a number of existing Java initiatives to provide a standard platform on which to build distributed applications in Java. This initiative, known as the Java 2 Enterprise Edition (J2EE) standard, builds on the Java 2 standard, and adds most of the important programming interfaces in an application server [21]. For example, J2EE includes the EJB specification for server-side components, together with the necessary application programming interfaces (APIs) required for building clients and connecting them to these server-side components (e.g., the Java Server Page (JSP) API for dynamically generating web pages, and the Java Naming and Directory Interface (JNDI) for locating and accessing distributed Java objects).

To fulfill the promise of making the development of component-based systems in Java easier, the J2EE augments this collection of standards and interfaces with [22]:

- A programming model for developing applications targeting the J2EE platform. This provides an outline approach to distributed application development, and highlights a number of key design heuristics for creating efficient, scalable solutions in Java.
- A compatibility test suite verifying J2EE platform implementations conform to the J2EE platform as defined by the collection of standards and APIs. This encourages portability of solutions across different vendor's implementations of the J2EE platform.
- A reference implementation that offers an operational definition of the J2EE platform. This demonstrates the capabilities of the J2EE platform, and supplies a base implementation for rapid prototyping of applications.

By collecting these elements together under a single umbrella, Sun aims to simplify the creation of distributed systems development for Java.

Chapter 7. Approaches to Component-Oriented Modeling

In the previous chapter we considered the core set of component technologies important **for any enterprise-scale component solution**. While the technology is a necessary element of any solution, on its own it is insufficient. A major part of any solution is the approach to be taken (i.e., the methods, techniques, processes, and heuristics) to apply the technology in context. These elements are essential for effective use of the technology.

In this regard, a number of questions must be addressed. In particular, how do we describe the behavior expected from each component, and arrive at an appropriate component architecture for an application? The answer lies in following the steps of a component-based modeling approach.

The goal of this chapter is to provide a high-level understanding of the major elements of component-based approaches to enterprise-scale solutions in the Internet age. Consequently, we describe the main steps and techniques of component-based modeling approaches, and compare component modeling techniques and their role in provisioning enterprise-scale solutions in the Internet age. More detailed studies of specific methods and approaches to component-based design are available elsewhere [1]–[4].

The primary focus of the RUP is derived from earlier work by Ivar Jacobson on the Objectory process [7]. Hence, the RUP is driven by a use-case-centered view of developing system requirements. An additional perspective has expanded this by emphasizing a set of models highlighting the proposed system's structure and architecture. This is referred to as the "4+1" view of a system due to the fact that it offers four views of a system, each offering insight into different aspects of a system's behavior. The synthesis of these views leads to a robust central software architecture for the system. As a result, the RUP provides a user-centered design approach concentrating on deriving flexible software architectures. The various perspectives are synchronized and refined by encouraging an iterative approach to development based on many of the techniques familiar from rapid application development, and exemplified in a number of spiral-based object-oriented approaches [8].

Components package(combine, bundle, wrap, file) the interfaces and classes shown on class diagrams.

8.2. Understand the Context

For this example we consider three activities to be primary in understanding the context:

- Requirements definition to obtain a clear statement of initial requirements;
- Use case modeling to identify current users and their activities;
- Business type modeling to create a representation of the main business roles and relationships in the domain of interest.

Requirements Definition

As with every software project, the early stages are taken up with attempting to obtain a collection of high-level statements that summarize the sponsor's requirements. Often these will be provided by the sponsor based on previous analysis and research. Other times this information must be extracted and confirmed based on meetings with management, employees, and customers.

For this illustration we shall assume that a statement of requirements has been provided for the project. These requirements cover a wide range of needs. Many involve the functional expectations for the system's operation and management, while others describe the quality and performance metrics for operation of the system, and define specific technical criteria restricting the solution's characteristics to ensure it is maintainable within the context of a wide range of existing systems. Other requirements may focus on effective management of the development project itself, such as the reporting criteria for the project, key milestones and dates, etc.

Typically, the requirements are documented in some textual form, perhaps supported by tools for assisting with the traceability of those requirements into design artifacts, implementation, and test scripts. More frequently, as in this illustration, the requirements are recorded in a textual document, and traceability is managed manually and recorded in a spreadsheet.

Once agreed upon, these requirements form the basis of future development. Ideally, they are testable statements used to build test cases and as acceptance criteria. At the very least, the project team is usually required during system acceptance testing to give some written or verbal mapping between these requirements statements and the implemented system.

Requirements rarely remain static throughout the life of a project, however. Over time, the needs of the users of the system evolve, and more appropriate solutions are defined due to a greater understanding of their needs. The initial requirements definition is used as the basis for further negotiation on the system's scope and functionality.

Use Case Modeling

Over the past few years it has been found that a great deal can be learned about the context for a future system by considering typical examples of its proposed use. This idea has been formalized with the concept of a use case. A use case is a description of a user's interaction with a computer system. A use case can be expressed as a goal (e.g., "withdrawing money from one account and depositing it in another") or as an interaction (e.g., "initiating a transaction to transfer money between two accounts"). In either situation, use cases are a well established technique for documenting the current or desired behavior of a system from a user's perspective.

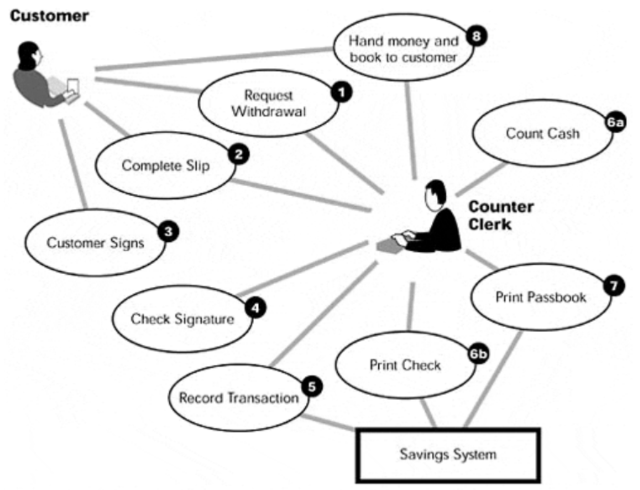
Some of the issues associated with use case modeling stem from deciding which interactions are valuable to model as use cases, understanding the relationships and dependencies among defined use cases, and documenting use cases at an appropriate granularity. For component-based approaches, the answers to these questions arise from a number of sources. These include:

- Extracting typical services from the statements of requirements. Sometimes these are explicitly described (e.g., "The system shall allow a teller to list the last five transactions for any customer" results in an obvious use case to allow a teller to list customer transactions). However, more often they are implicitly inferred from the external behavior required of the system (e.g., "Checks for amounts in excess of $10,000 must be countersigned by a supervisor" implies there are different situations to be described for cashing checks of different values).
- Examining the results of a previous business process improvement excercise, or business process reengineering (BPR) project. Described processes and activities are often suitable candidates for use cases. Typically, a direct mapping exists from individual workflows in a BPR model to a first-cut set of use cases.
- Identifying key business events in the domain of interest to see which activities occur as a result of each event. The events here are external actions that occur periodically (e.g., "At 5pm each evening a summary of the day's transactions is produced"), or as a result of some stimulus (e.g., "When a teller suspects that a check may be fraudulent, they immediately inform a supervisor"). Each such event is a potential use case.

As an illustration of the use cases that will be defined, Figure 8.2 shows a use case for a customer making a cash withdrawal from his or her account. Note that the use case involves a number of actors (customer and teller) carrying out sequenced actions with each other and the system (Savings System).

The use case technique has a number of benefits. In particular, the defined use cases are often used to confirm understanding of the requirements documents, and to engage future users of the system in a dialog to ensure there is a common understanding of how the system will operate.

Figure 8.2. An example of a use case.


Customer
Hand money and book to customer
8
Request Withdrawal
1
Count Cash
6a
Complete Slip
2
Counter Clerk
Customer Signs
3
Print Passbook
7
Check Signature
4
Print Check
6b
Record Transaction
5
Savings System