4504_CH01 Page 1 Friday, September 15, 2000 2:22 PM

Chapter 1

Component Systems

Components aren't rocket science. Try explaining the basic principles of component-based development to someone not involved in the information technology (IT) industry and they're likely to be somewhat underwhelmed. Despite this apparent simplicity, there seems to be widespread misunderstanding and confusion about the concepts that underlie components and about what makes them special. In this chapter we take a tour around some of the major topics that distinguish component systems from systems built using other approaches. We also establish some terminology and concepts, and set out our approach to structuring component systems. The goal is to get acquainted with the basic mindset needed so that you are well armed before we plunge into the Unified Modeling Language (UML) and our case study.

1.1 Component Goals

Component systems adhere to the principle of divide and conquer for managing complexity—break a large problem down into smaller pieces and solve those smaller pieces, then build up more elaborate solutions from simpler foundations. Easy. To many this sounds like the structured methods that have been followed for years, so what's different?

The principal difference is that components follow the object principle of combining functions and related data into a single unit. Traditional structured approaches have tended to focus mainly on functional decomposition and have maintained a strong distinction between data and function.

4504_CH01 Page 2 Friday, September 15, 2000 2:22 PM

There's one thing that's worth getting clear early on—what major challenge is the component approach to system development addressing? For us that challenge is managing change. This means building for change in the first place by placing primary emphasis during architecture and design on the dependencies between the components, and the management of those dependencies. The purpose of the individual components is clearly important but is in many ways a secondary concern.

This may surprise some people. Many think the primary objective of components is reuse. They want to design something once and use it over and over again in different contexts, thereby realizing large productivity gains, taking advantage of best-in-class solutions, the consequent improved quality, and so forth. These are admirable objectives, but the main driver today is that things keep changing, and often—as with business-to-business electronic commerce—there is no longer any hope that centralized control can be exerted. In such an environment one of the primary objectives of a component is that it must be easily replaceable—either by a completely different implementation of the same functions or by an upgraded version of the current implementation. This places the emphasis on the architecture of the system, on being able to manage the total system, as its various components evolve and its requirements change, rather than seeking to ensure that individual components are reusable by multiple component systems.

We're focusing on the whole, rather than the parts.

1.2 Component Principles

Components, as found in the component technologies we are considering here, are units of software structured according to some specific principles. The fundamental principles they adhere to are actually the fundamental principles that underpin object technology:

1. **Unification of data and function:** A software object consists of data values (or state) and the functions that process those data. This natural colocation of dependencies between function and data improves cohesion.

2. **Encapsulation:** The client[1] of a software object is insulated from how that software object's data is stored or how its functions are implemented. We say that the client depends on the object specification, but not its implementation. This is a very important separation of concerns and is key to managing dependencies between software and reducing coupling.
3. **Identity:** Each software object has a unique identity, regardless of state.

Components extend these object principles by elaborating the notion of an object specification with an explicit representation of specification dependency called an *interface*. This is an important level of indirection between the client of an object and the capabilities of that object. The specification of the total capabilities of an object may be split across several interfaces.

To summarize: Component-based development is different from previous approaches in its separation of component specification from implementation, and in the division of component specifications into interfaces.

Splitting component specifications into one or more interfaces means that the intercomponent dependencies can be restricted to individual interfaces, rather than encompass the whole component specification. This reduces the impact of change because one consumed component may replace another even if it has a different specification, as long as its specification includes the same interfaces the consuming components require.

These characteristics allow a component to be upgraded or replaced with minimal impact on the clients of that component.

Figure 1.1 shows an existing component being replaced by a new one. An existing client will use established interfaces (characterized by IX) and will be unaffected—new components can be plugged in to existing clients if they offer the old interfaces. They may also provide new functionality (IX+) for new clients.

Clients may also be flexible in their requirements of components. They may require certain interfaces as a minimum, but will leverage additional or newer interfaces if they are available.

1. *Client* here means consuming or calling software.

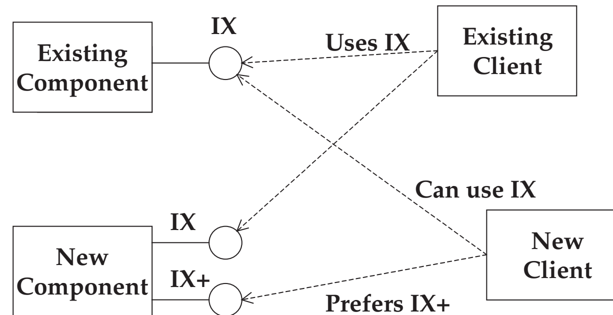


Figure 1.1 Interface dependency supports component replacement

1.3 Component Forms

So, what exactly is a component? This is actually a surprisingly difficult question to answer simply, even when you restrict your definition of components to current technologies, because the word *component* is used informally to mean a variety of things, each with apparent equal validity.

A better question, and one we *can* answer simply, is, What features does a component have and how does our view of it change during a project life cycle? From requirements and specification, through design and provisioning, to assembly, deployment, and runtime, the characteristics we want from a component vary. We can identify a number of component forms, each form reflecting some aspect of a component during the development life cycle.

So, in fact, we won't define component at all; instead we define the many forms a component can take.

- Being able to assemble applications from components means that components must conform to some sort of environment standard—they form part of a component kit. Just like buying an off-the-shelf part in any other domain (such as computer hardware, automobiles, or washing machines) a component will only plug in if it conforms to some laid down set of base standards. So, the shape of the plug-in piece is important. Let's call this the **Component Standard**. Enterprise JavaBeans (EJB) and Microsoft's COM+ are examples of such standards. Some

4504_CH01 Page 5 Friday, September 15, 2000 2:22 PM

large organizations have defined their own. Provided you have one and it meets your needs it doesn't much matter what it is, but we claim that it is meaningless to talk about components unless you have a standard—or possibly more than one—in mind.

- When you go looking for a component to plug in, having the right shape plug is certainly a good start, but knowing what that part does is pretty important too. Consider the fuse that's probably sitting in the power supply of your computer: It's no use buying a 5-amp fuse when you need a 15-amp fuse. They both slot in nicely, but one will blow when it shouldn't, or conversely one won't blow when it should. So, the specification of what a component does must also be part of a valid definition—we need a clear **Component Specification**.
- A major part of a component specification is the definition of **Component Interfaces**, or just **Interface** for short.
- The specification of the component is more important, from an assembly perspective, than the way that specification is realized or implemented. It should be possible to replace one component with another (of an equivalent specification) without affecting the assembly. For example, you may want to be able to replace one 15-amp fuse with another from a different manufacturer. What matters from an assembly point of view is the interdependency between the parts, not the way those parts work. The clear separation of component specification from **Component Implementation** is therefore another important characteristic of a component. The assembly itself should only depend on the specification. If there is any dependency on the implementation then the ability to replace that piece easily will be lost.

The lack of dependency on implementation means that software components can be implemented in any programming language using any data storage mechanism. In particular, this means that existing software, which may not initially meet all the requirements of a component, can be updated, sometimes very simply, so that it conforms to the component standard and becomes a valid component.

4504_CH01 Page 6 Friday, September 15, 2000 2:22 PM

- It is the component implementations that are deployed onto machines. Each time we install a component implementation we create an **Installed Component** (i.e., an installed version of the component, known to the environment).
- We also need to consider the existence of state or content of the component at runtime. While the software services provided by a component are important, so is the information managed by that component. When replacing a component it is not usually enough to substitute equivalent services—the information managed must also be equivalent. Imagine replacing a component managing millions of customers to find that the new component provided equivalent customer management services but had no knowledge of the customers! Similarly, we might want to manage European and U.S. customers separately, using two different instances of the same component. Since we are now referring to services and state together, a key object principle, we call each instance created from an installed component a **Component Object**. It is these component objects that have an identity. Only component objects can actually *do* anything.

It would be useful if UML directly supported all these distinctions about components, but unfortunately it doesn't. In Chapter 3 we explain how we can adapt UML for use in modeling components.

We've introduced quite a number of important distinctions that are needed when modeling components. These are summarized in Figure 1.2 and Table 1.1.

1.3.1 Example: Microsoft Word

As an example, consider Microsoft Word and its components. Word comprises many components but two of the most obvious are those representing the application itself and the documents you open or create. Referring to Figure 1.3, we assume that back at Microsoft HQ there exist specifications of these two components, but these aren't supplied on the CD. Instead we get an executable file called winword.exe that packages the Component Implementations. These are implementations specific to the Component Standard we are using, for example COM.

4504_CH01 Page 7 Friday, September 15, 2000 2:22 PM

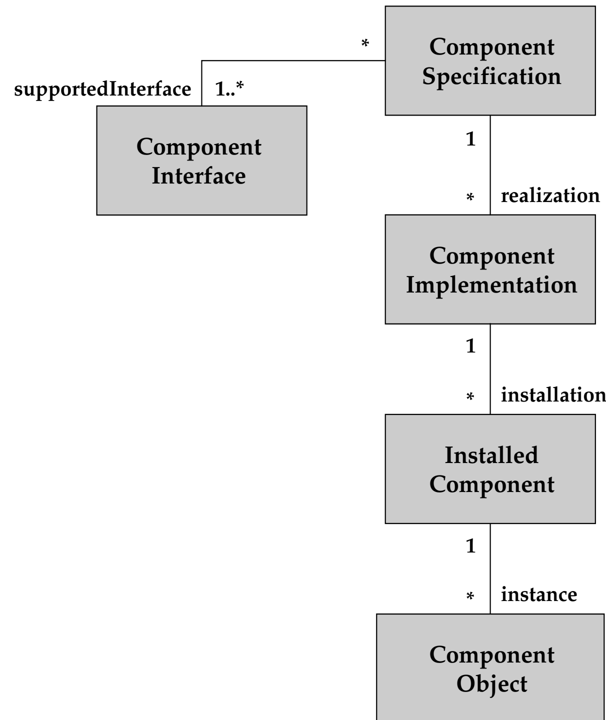


Figure 1.2 Component forms

Simplifying somewhat, installing Word involves copying the physical file winword.exe into a known location and registering its contents with the runtime environment (in this case COM). This creates two Installed Components—components known to COM. Running the Word application initially creates two Component Objects: one representing the application object itself, which acts as a frame, and another as a default new document object. Further document objects can be created using the File/New command on the application object.

1.3.2 What a Component Isn't

We hope you've followed the logic so far. But to rule out any possible confusion we'll take a moment to explain some things that a component isn't.

4504_CH01 Page 8 Friday, September 15, 2000 2:22 PM

Table 1.1 Component form descriptions

Component Form	Description
Component Specification	The specification of a unit of software that describes the behavior of a set of Component Objects and defines a unit of implementation. *Behavior* is defined as a set of Interfaces. A Component Specification is realized as a Component Implementation.
Component Interface	A definition of a set of behaviors that can be offered by a Component Object.
Component Implementation	A realization of a Component Specification, which is independently deployable. This means it can be installed and replaced independently of other components. It does not mean that it is independent of other components—it may have many dependencies. It does not necessarily mean that it is a single physical item, such as a single file.
Installed Component	An installed (or deployed) copy of a Component Implementation. A Component Implementation is deployed by registering it with the runtime environment. This enables the runtime environment to identify the Installed Component to use when creating an instance of the component, or when running one of its operations.
Component Object	An instance of an Installed Component. A runtime concept. An object with its own data and a unique identity. The thing that performs the implemented behavior. An Installed Component may have multiple Component Objects (which require explicit identification) or a single one (which may be implicit).

A component isn't an object, not in the sense of simply being an object in a Java or C++ program. It's true that at runtime we have Component Objects that exhibit most of the characteristics of programming language objects, but they exist in the context of a Component Standard, such as provided by an EJB server. Component Objects can only exist in such a context.

A component isn't a service, although one of the things you can do with components is build **Service-Based Architectures**, where each Component Object provides a specific function, using specific data. But components can equally well be used to abstract data, as you will see later in this book.

4504_CH01 Page 9 Friday, September 15, 2000 2:22 PM

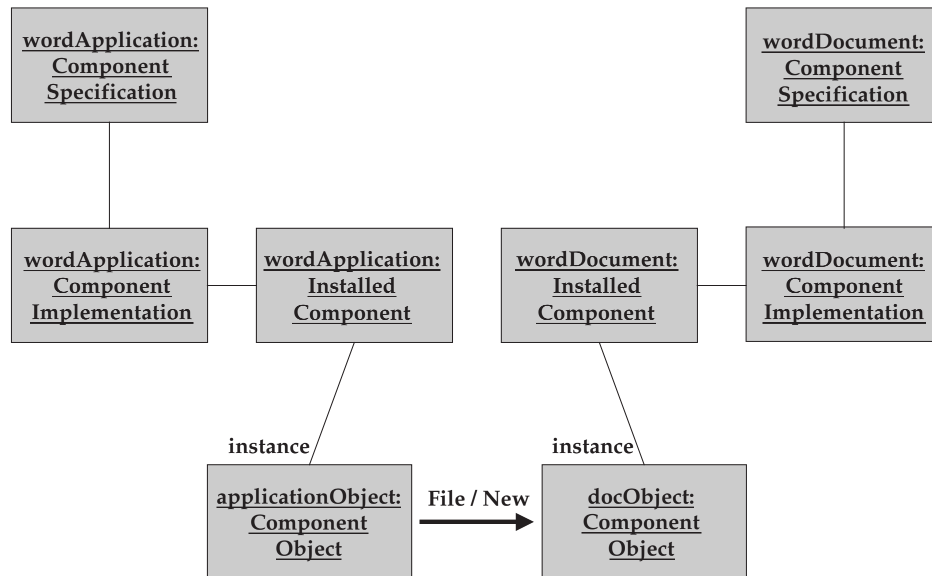


Figure 1.3 Microsoft Word

1.4 Component and System Architectures

Architecture is one of those words that has lost some of its meaning over time. There are so many types of architecture in common usage we need first to define what we mean by the term. Here we are going to describe **System Architectures** and **Component Architectures**.

We define system architecture to be

> *the structure of the pieces that make up a complete software installation, including the responsibilities of these pieces, their interconnection, and possibly even the appropriate technology.*

We define a component architecture as

> *a set of application-level software components, their structural relationships, and their behavioral dependencies.*

So we are following the trend of using architecture to mean the overall structure of a system or family of systems. Whatever word you choose for

4504_CH01 Page 10 Friday, September 15, 2000 2:22 PM

this, it is undeniably important to understand the big picture: what pieces you've got, how they fit together, and what the impact of change might be. That is the value of architecture.

1.4.1 System Architectures

Our aim in this book is to provide advice, guidance, and examples for modeling enterprise-scale component systems. This means we're considering N-tier distributed architectures, linking corporate databases, off-the-shelf packages, and legacy systems, through business-process-specific application software to web-based user interfaces. A typical system architecture for the kinds of systems we have in mind is shown in Figure 1.4.

Understanding the system architecture is important because it tells us the overall shape of the final system and explains how we will use various technologies to assemble the system we need.

Architecture Layers

We want to use components for different purposes and to keep different concerns separate. Our general approach is to identify different layers into which

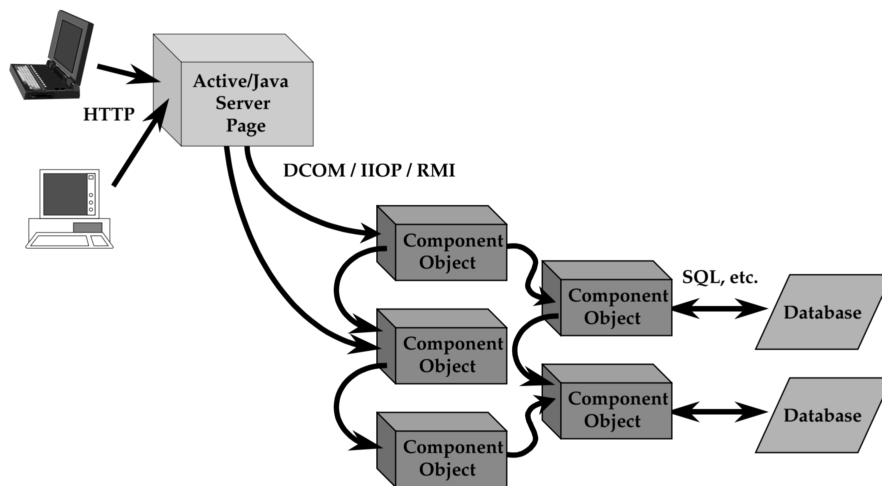


Figure 1.4 Typical system architecture

4504_CH01 Page 11 Friday, September 15, 2000 2:22 PM

components can fit (see Figure 1.5). This is useful because it allows us to reason about the purpose of each software unit we put into the application.

The layers are as follows:

- **User interface**—the presentation of information to the users and the capture of their inputs. External systems can be users, too. Although there might be things called components in this layer (e.g., user interface widgets), they won't be the kind of components we're considering in this book.
- **User dialog**—management of the user's dialog in a session. Once again, we aren't, in this book, dealing with components in this layer.
- **System services**—the external representation of the system, providing access to the services of the system. This layer acts as a facade for the layer below, providing a context within which the more general business services are used to meet the needs of this particular system.
- **Business services**—the implementation of core business information, rules, and transformations. These are often reused across several systems.

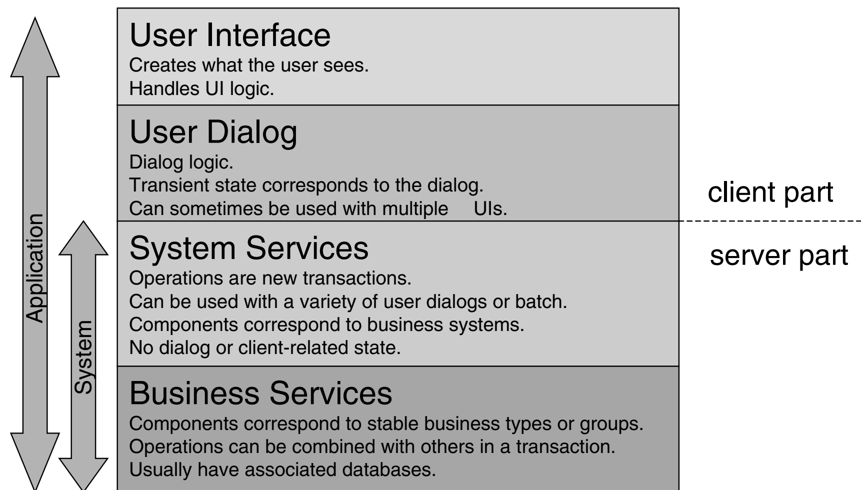


Figure 1.5 Architectural layers

We consider the bottom two layers to form the **System,** which we define to be user interface (UI) independent. When a UI is connected to the system we have an **Application**. Many different UIs may be connected simultaneously to a single system. Figure 1.6 shows an example of the component objects we might find in the system layers, using the specific technology of the Java 2 Enterprise Edition.

We aren't making hard and fast rules about the use of these layers. For example, we don't insist that communication is always between adjacent layers. A common case is when the user dialog layer directly accesses the business services layer. In some systems the system services layer might be completely empty, implying that the underlying business service components require no management to allow them to function in the context of this system. At the other extreme, we might instigate a rule that all access to the business service component objects is always via a single system service component object (sometimes called a "radial architecture") because we want to keep the lower-level component objects isolated, hopefully improving the interchangeability and reuse potential of each component.

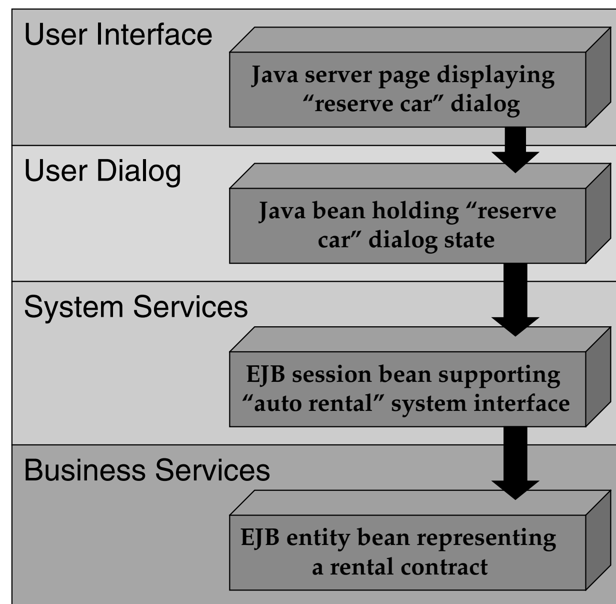


Figure 1.6 Example components in the layers

1.4.2 Component Architectures

Components of some form or another might be found in all four layers of the system architecture. However, this book is concerned with the server parts of the application, so we will focus on modeling the components that live in the system services and business services layers.

We said before that a **Component Architecture** is a set of application-level software components, their structural relationships, and their behavioral dependencies. This is a logical definition and is independent of the technology level on which it will be deployed. A component architecture may apply to a single application or to a wider context, such as a set of applications serving a particular business process area. Creating this logical view of the components allows us to understand how tightly or loosely coupled our system is, and to reason about the effects of modifying or replacing a component.

By structural relationships we mean associations and inheritance between component specifications and component interfaces, and composition relationships between components. By behavioral dependencies we mean dependency relationships between components and other components, between components and interfaces, and between interfaces (see Figure 1.7).[2]

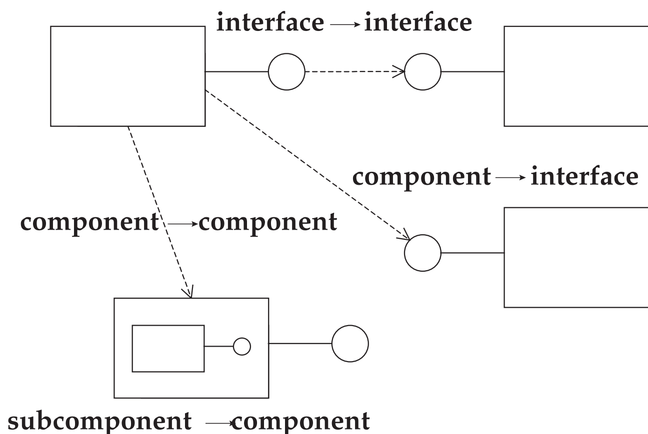


Figure 1.7 Dependencies in component architectures

2. Subcomponents are discussed in Chapter 8.

4504_CH01 Page 14 Friday, September 15, 2000 2:22 PM

Component Specification Architectures

It is possible to create component architectures that focus on either component specifications, component implementations, or component objects. A **Component Specification Architecture** diagram contains only component specifications and interfaces. All dependencies take the form of an interface or a component specification depending on an interface. The important rule for specification architectures is this:

> *Any dependency emanating from a component specification is part of the definition of that component specification and must be adhered to by all implementations.*

The dependency acts as a constraint on all implementations of the specification: Any implementation must exhibit this dependency. Typically the dependencies in a component specification architecture are derivable from more detailed parts of the specification, such as interaction diagrams.

Do we really need to specify, at an architectural level, all these dependencies? Can't we leave many of the decisions to the component designers? In an isolated development project many of the component dependencies may be left as a design decision, but for larger-scale, enterprise-wide application development, the component specification architecture is a critical aspect of the wider IT architecture. It is not up to individual application development teams to pick and choose which components they wish to reuse—it is often an architectural constraint placed on them by a centralized architecture team.

By being explicit about the specification dependencies, a component specification architecture can be used as a mechanism for ensuring that broader standards, policies, and approaches are followed. For example, an individual application may find the performance overhead of an accounting component unnecessary. But if its use is made standard policy by enforcing the rule that every component must use its services, then broader benefits can follow, such as standardized accounting or reporting across all applications.

Component Implementation Architectures

The **Component Implementation Architecture** shows the actual dependencies that exist between particular component implementations. These will be the union of the dependencies that are specified and those intro-

4504_CH01 Page 15 Friday, September 15, 2000 2:22 PM

duced by the component realizers: Component implementations may interact with components not mentioned in the specification, and they may have additional interactions with components that are mentioned.

Component Object Architectures

We hear a lot about component reuse in the press. This may be a straightforward goal for stateless components, but for stateful components it is also important to understand, and be precise about, which component objects are being used.

Consider the situation where a component specification architecture defines that both the order manager component and the stock control component must use the product manager component (see Figure 1.8).

Although the intent, and even the impression, may be that there is a single set of product information accessed by the two calling components, the model does not state that this must be the case. To be precise we need to use a **Component Object Architecture**, specifying the instances of components that will be accessed.

In Figure 1.9, two alternative component object architectures are shown, each conforming to the component specification architecture in Figure 1.8. In Figure 1.9(a) the order manager and the stock control component objects share a common product manager component object. In Figure 1.9(b) each has its own instance of the product manager component and manages a distinct set of product information.

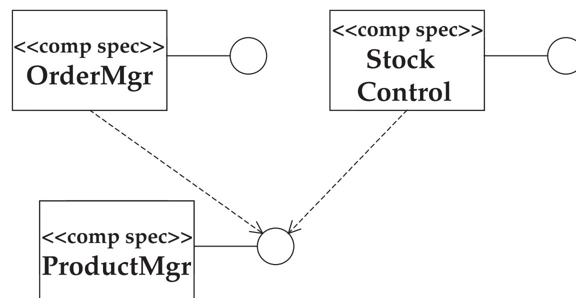


Figure 1.8 Component specification architecture

4504_CH01 Page 16 Friday, September 15, 2000 2:22 PM

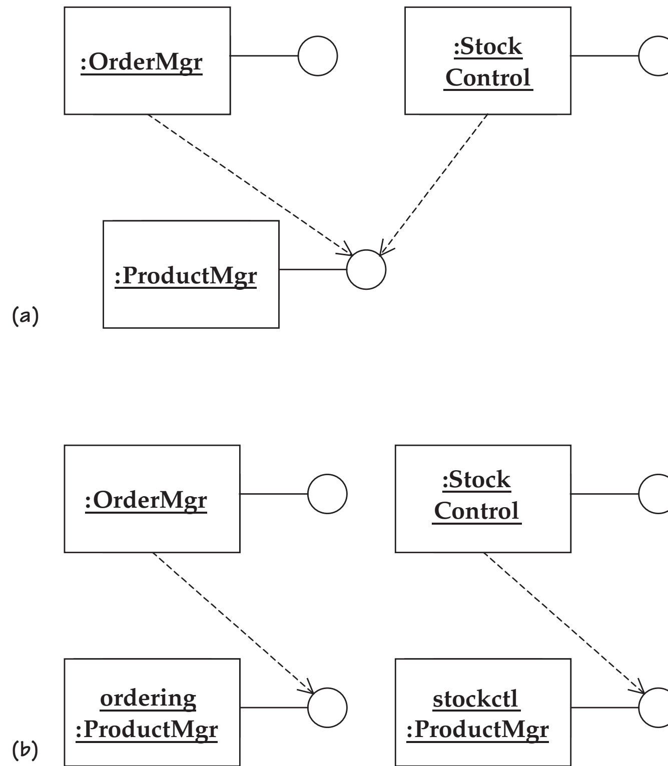


Figure 1.9 Alternative component object architectures

1.5 Specifying Contracts

Designing good component-based software can be seen, at the macroscopic or architectural level, as a dependency management problem. We are seeking to ensure that pieces of software, developed at different times by different people, possibly from different organizations, can work together successfully. How can we do this?

Well, software development is a relatively new discipline in the grand scheme of things. Software engineers haven't had long to work it all out, so let's draw on the experience of other human endeavors that have analogous issues and that have stood the test of time. We might learn some-

thing useful. If we think of a component as a piece of software that provides some sort of service, and requires services of others, we can usefully make an analogy between components and companies.

Companies are entities that provide services to their customers (which may be other companies, organizations, or individuals) and that often depend on the services of other companies. How do companies manage their relationships in the real world? They do it through contracts. A contract is a formal agreement between two or more parties. It describes (or specifies) the detail of the agreement in an unambiguous form. This involves stating the responsibilities or obligations of each party—what each party will do provided the other parties do what they say they will do. It does not state how they will do it, simply that they will. Importantly, it also states what will happen if the parties do not do what they say—if they fail to provide the agreed service.

The analogous **Design by Contract** has proved very useful in software design. It was developed extensively in the object-oriented domain by Bertrand Meyer [Meyer00] and applied to the component world by Catalysis [D'Souza99]. Building on this work, we can apply design by contract to UML and today's component technologies.

We distinguish two different types of contract:

1. **Usage—**the contract between a component object's interface and its clients
2. **Realization—**the contract between a component specification and its implementation

These different aspects are shown in Figure 1.10.

It's important to distinguish these contracts because they correspond to the roles often played in the development of large component systems: The people who build components are often not the same people who use them. The contents and purpose of the contracts are really quite different.

The specification of a component is composed of many parts, not all of them relevant to the client. Defining which interfaces the component must offer is just part of the whole component specification. The interface defines everything the client needs to know, but no more than that. An interface doesn't, for example, specify how implementations of this interface must interact with other components to fulfil their responsibilities; that is defined in the component specification itself. The interface specification may imply

4504_CH01 Page 18 Friday, September 15, 2000 2:22 PM

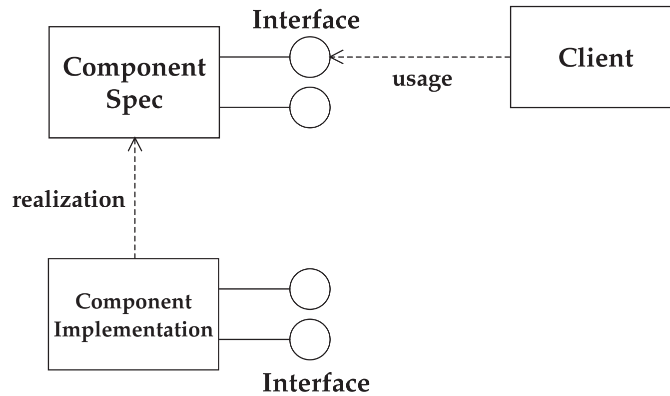


Figure 1.10 Different types of contracts

interactions but it does not specify how they happen. Other parts of the component specification do this.

The primary reason for keeping these things separate is to facilitate change: A change to the realization constraints does not constitute a change to the usage contract, and hence does not affect clients. This is important because it gives us the ability to change specifications that affect realization without having to revalidate all client usage.

1.5.1 Usage Contracts

A **Usage Contract** (see Figure 1.11) describes the relationship between a component object's interface and a client, and is specified in the form of an interface. The client is left unspecified because we can't predict who will use an interface in the future. The specification of an interface includes the following:

- **Operations**—a list of operations that the interface provides, including their signatures and definitions

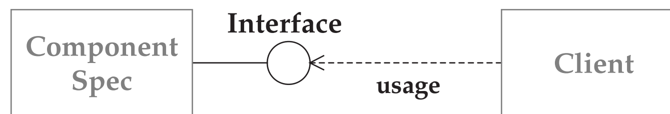


Figure 1.11 Usage contract

4504_CH01 Page 19 Friday, September 15, 2000 2:22 PM

- **Information model**—the abstract definition of any information or state that is retained between client requests by an object supporting the interface, and any constraints on that information

Each operation is treated as a fine-grained contract in its own right. It defines its inputs, its outputs, the relationship between them, and the conditions under which they apply. The reason for grouping operations into larger-grained contractual units (i.e., interfaces) is because their pragmatic usage needs the existence of other operations and because they are rarely used independently of those other operations.

As well as a signature, each operation is defined by the following:

- **Precondition**—a definition of the situations under which the postcondition will apply
- **Postcondition**—a description of the effects of that operation on its parameters and the information model

The onus is on the client to ensure that the precondition of an operation is true before making a call. By agreeing to the contract, it accepts that if the precondition isn't met the expected behavior might not happen. For its part the supplier promises to satisfy the postcondition if the precondition was met. It's important to understand that if the precondition isn't met, the result of the operation is undefined—anything could happen.

Consider an order management application (see Table 1.2) with an interface IOrder defining the capabilities of an individual Order, with an operation createOrderLine( ) which takes as input a product-type identifier and a quantity. Here's the deal: The client ensures that the type of product being ordered is valid and that at least one is being ordered, while the supplier (the object supporting IOrder) makes sure an order line is created.

Using pre- and postconditions doesn't automatically imply a high degree of precision, but precise specifications are very valuable in defining components

Table 1.2 Order line creation contract

Party	Responsibility
Client	Product type exists and quantity > 0
IOrder	Order line created

4504_CH01 Page 20 Friday, September 15, 2000 2:22 PM

that can be replaced painlessly. Typically, much of the work involved in specifying interfaces centers around making the pre- and postconditions precise, often using a formal language. We explain this more fully in Chapter 7.

1.5.2 Realization Contracts

A usage contract is a runtime contract, but a **Realization Contract** is a designtime contract. It is a contract between a component specification and a component implementation, and must be adhered to by the person who is creating the implementation: the realizer. The realization contract is embodied in the component specification itself (see Figure 1.12).

While an interface defines a set of behavior, a component specification defines an implementation and deployment boundary. By specifying a set of interfaces that must be supported, a component specification defines the total sum of capabilities of any object created at runtime from the implementation. The component specification can also define how the implementation must interact with other components. We may want to instruct any implementer of the component specification that he or she must include particular interactions with other components as part of the implementation of operations. Knowledge of such interactions does not form part of the interface specification and so cannot be directly known by the client. We define these interaction constraints either procedurally using collaboration diagrams, or declaratively using constraints between the information models of the supported and used interfaces.

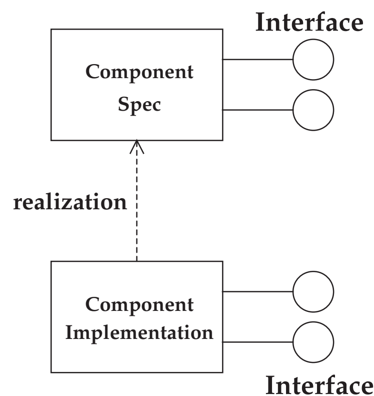


Figure 1.12 Realizing a component specification

4504_CH01 Page 21 Friday, September 15, 2000 2:22 PM

Finally, a component specification can specify how one supported interface must correspond to another. As noted, an interface specification contains an information model: an abstract model of the information retained by the component between operations. When a component supports two or more interfaces, elements in that information model of one interface often correspond directly to elements in the information models of other supported interfaces, and must always have the same value. Similarly, an element in the information model of a supported interface may correspond to an element in a used interface. We specify such constraints between interface information models as part of the component specification.

All this information, together with the specifications of the supported interfaces, constitutes the realization contract.

1.5.3 Interfaces versus Component Specifications

Although at first sight it may not be apparent why we need both interface and component specification as separate concepts, they perform quite distinct functions. The component specification forms the contract with the component realizer, assembler, and tester. A component specification scopes the implementation unit, defines the encapsulation boundary, and consequently determines the granularity of replaceability in the system. The interface forms the contract with the component client. It tells the client what to expect. Table 1.3 summarizes the differences between an interface and a component specification.

Table 1.3 Interface versus component specification

Interface	Component Specification
A list of operations	A list of supported interfaces
Defines an underlying logical information model	Defines the relationships between the information models of different interfaces
Represents the contract with the client	Represents the contract with the implementer
Specifies how operations affect or rely on the information model	Defines the implementation and runtime unit
Describes local effects only	Specifies how operations must be implemented in terms of usage of other interfaces

4504_CH01 Page 22 Friday, September 15, 2000 2:22 PM

1.6 Model Levels

In this book we talk a lot about models, mostly using UML as our modeling language. We need to be clear on the purpose of these models.

Any model of something is a perspective or view—it includes and emphasizes some things and excludes others. That's the value of a model. For example, a map of a country could highlight political boundaries, geography, geology, transport, industry, population, economics, ecology, and so on. If you draw a map for someone and you want them to understand it, they'd better know what all the lines mean and what the perspective is. That's what the map key is for. It's the same with models in the software domain. How can we ensure that two people both understand the meaning of a box and line diagram in the same way?

UML provides a standard language that can be used in many ways. In particular, it is useful to distinguish three quite different semantic levels of model that are frequently drawn using UML [Cook94] [Fowler99]:

1. **Conceptual models**—software-independent models that identify the concepts in the domain being studied
2. **Specification models**—software-specific models that define the specification of the software, not its implementation details; models of the "outsides" of components
3. **Implementation models**—models that detail the implementation design of the software; models of the "insides" of components

In fact UML contains elements that correspond to different levels of model. For example, the class concept has stereotypes for «type» and «implementationClass»; but more about that in Chapter 3.

Since we're interested in how to model components and their dependencies, we'll be dealing mainly with specification models—the outsides of components. We'll be showing how UML can be applied to model these outsides in a precise manner. We won't concern ourselves with the insides of components (their internal design and implementation) other than their implementation dependencies on other components. A component may be implemented in any language, from COBOL to Java. That's one of the useful things about them.

4504_CH01 Page 23 Friday, September 15, 2000 2:22 PM

What's more, we'll be focusing on applying UML to models of software applications, not to business or technology models. Application models focus on the structure of an application and usually exclude considerations of the underlying technology on which that application is based. For example, an application model may describe dependencies between a product management component, an order management component, and a billing component, but not show what distributed component standard they conform to or what language they are implemented in.

1.7 Summary

This book is concerned with specification of business systems.

Components adopt object principles: unification of data and function, encapsulation, and identity. Components extend these principles by strengthening the role of the interface and by adding a separate notion of component specification. Components must conform to a component standard.

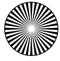
The systems we are targeting with this book can be structured into four layers:

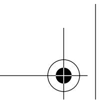
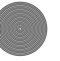
- User interface
- User dialog
- System services
- Business services

These layers and the dependencies between them constitute a system architecture.

A component architecture is concerned with the structure and dependencies between components in the system services and business services layers. The component specification architecture describes the constraints on the implementation and assembly of components to form the system. The component implementation architecture describes the actual dependencies between specific component implementations.

The component object architecture describes the runtime situation: what instances of components there are and the dependencies between these instances.



4504_CH01 Page 24 Friday, September 15, 2000 2:22 PM



The component specification defines what is to be built and what units will exist at runtime. The component specification defines the set of interfaces supported and any constraints on how they are to be implemented. It is the contract between the specifier and the realizer: the realization contract.

An interface specifies the operations it contains. Each operation specification is a contract between the invoker and the component providing the operation. Interfaces are usage contracts.