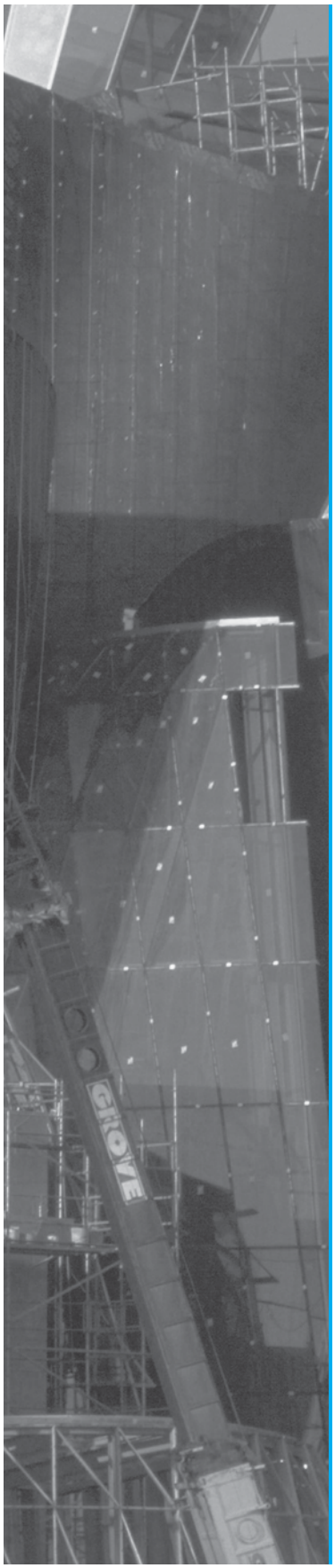

17

Component-based software engineering

Objectives

The objective of this chapter is to describe an approach to software reuse based on the composition of reusable, standardized components. When you have read this chapter you will:

- know that component-based software engineering is concerned with developing standardized components based on a component model, and composing these into application systems;
- understand what is meant by a component and a component model;
- know the principal activities in the CBSE process for reuse and the CBSE process with reuse;
- understand some of the difficulties and problems that arise during the process of component composition.

Contents

As I explained in Chapter 16, many new business systems are now developed by configuring off-the-shelf systems. However, when a company cannot use an off-the-shelf system because it does not meet their requirements, the software they need has to be specially developed. For custom software, component-based software engineering is an effective, reuse-oriented way to develop new enterprise systems.

Component-based software engineering (CBSE) emerged in the late 1990s as an approach to software systems development based on reusing software components. Its creation was motivated by designers' frustration that object-oriented development had not led to extensive reuse, as had been originally suggested. Single object classes were too detailed and specific, and often had to be bound with an application at compile time. You had to have detailed knowledge of the classes to use them, and this usually meant that you had to have the component source code. This meant that selling or distributing objects as individual reusable components was practically impossible.

Components are higher-level abstractions than objects and are defined by their interfaces. They are usually larger than individual objects and all implementation details are hidden from other components. CBSE is the process of defining, implementing, and integrating or composing loosely coupled, independent components into systems. It has become as an important software development approach because software systems are becoming larger and more complex. Customers are demanding more dependable software that is delivered and deployed more quickly. The only way that we can cope with complexity and deliver better software more quickly is to reuse rather than reimplement software components.

The essentials of component-based software engineering are:

1. Independent components that are completely specified by their interfaces. There should be a clear separation between the component interface and its implementation. This means that one implementation of a component can be replaced by another, without changing other parts of the system.
2. Component standards that facilitate the integration of components. These standards are embodied in a component model. They define, at the very minimum, how component interfaces should be specified and how components communicate. Some models go much further and define interfaces that should be implemented by all conformant components. If components conform to standards, then their operation is independent of their programming language. Components written in different languages can be integrated into the same system.
3. Middleware that provides software support for component integration. To make independent, distributed components work together, you need middleware support that handles component communications. Middleware for component support handles low-level issues efficiently and allows you to focus on application-related problems. In addition, middleware for component support may provide support for resource allocation, transaction management, security, and concurrency.
4. A development process that is geared to component-based software engineering. You need a development process that allows requirements to evolve, depending

Problems with CBSE

CBSE is now a mainstream approach to software engineering—it is a good way to build systems. However, when used as an approach to reuse, problems include component trustworthiness, component certification, requirements compromises, and predicting the properties of components, especially when they are integrated with other components.

http://www.SoftwareEngineering-9.com/Web/CBSE/problems.html

on the functionality of available components. I discuss CBSE development processes in Section 17.2.

Component-based development embodies good software engineering practice. It makes sense to design a system using components, even if you have to develop rather than reuse these components. Underlying CBSE are sound design principles that support the construction of understandable and maintainable software:

1. Components are independent so they do not interfere with each other's operation. Implementation details are hidden. The component's implementation can be changed without affecting the rest of the system.
2. Components communicate through well-defined interfaces. If these interfaces are maintained, one component can be replaced by another, which provides additional or enhanced functionality.
3. Component infrastructures offer a range of standard services that can be used in application systems. This reduces the amount of new code that has to be developed.

The initial motivation for CBSE was the need to support both reuse and distributed software engineering. A component was seen as an element of a software system that could be accessed, using a remote procedure call mechanism, by other components running on separate computers. Each system that reused a component had to incorporate its own copy of that component. This idea of a component extended the notion of distributed objects, as defined in distributed systems models such as the CORBA specification (Pope, 1997). Several different protocols and standards have been developed to support this view of a component, such as Sun's Enterprise Java Beans (EJB), Microsoft's COM and .NET, and CORBA's CCM (Lau and Wang, 2007).

In practice, these multiple standards have hindered the uptake of CBSE. It was impossible for components developed using different approaches to work together. Components that are developed for different platforms, such as .NET or J2EE, cannot interoperate. Furthermore, the standards and protocols proposed were complex and difficult to understand. This was also a barrier to their adoption.

In response to these problems, the notion of a component as a service was developed, and standards were proposed to support service-oriented software engineering.

The most significant difference between a component as a service and the original notion of a component is that services are stand-alone entities that are external to a program using them. When you build a service-oriented system, you reference the external service rather than including a copy of that service in your system.

Service-oriented software engineering, which I discuss in Chapter 19, is therefore a type of component-based software engineering. It uses a simpler notion of a component than that originally proposed in CBSE. It has been driven, from the outset, by standards. In situations where COTS-based reuse is impractical, service-oriented CBSE is becoming the dominant approach for the development of business systems.

17.1 Components and component models

There is general agreement in the CBSE community that a component is an independent software unit that can be composed with other components to create a software system. Beyond that, however, people have proposed varying definitions of a software component. Councill and Heineman (2001) define a component as:

> *"A software element that conforms to a standard component model and can be independently deployed and composed without modification according to a composition standard."*

This definition is essentially based on standards so that a software unit that conforms to these standards is a component. Szyperski (2002), however, does not mention standards in his definition of a component but focuses instead on the key characteristics of components:

> *"A software component is a unit of composition with contractually-specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

Both of these definitions are based on the notion of a component as an element that is included in a system, rather than a service that is referenced by the system. However, they are also compatible with the idea of a service as a component.

Szyperski also states that a component has no externally observable state. This means that copies of components are indistinguishable. However, some component models, such as the Enterprise Java Beans model, allow stateful components, so these do not correspond with Szyperski's definition. Although stateless components are certainly simpler to use, there are some systems where stateful components are more convenient and reduce system complexity.

What the above definitions have in common is that they agree that components are independent, and that they are the fundamental unit of composition in a system. In my view, a better definition of a component can be derived by combining these

Component Characteristic	Description
Standardized	Component standardization means that a component used in a CBSE process has to conform to a standard component model. This model may define component interfaces, component metadata, documentation, composition, and deployment.
Independent	A component should be independent—it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a 'requires' interface specification.
Composable	For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself, such as its methods and attributes.
Deployable	To be deployable, a component has to be self-contained. It must be able to operate as a stand-alone entity on a component platform that provides an implementation of the component model. This usually means that the component is binary and does not have to be compiled before it is deployed. If a component is implemented as a service, it does not have to be deployed by a user of a component. Rather, it is deployed by the service provider.
Documented	Components have to be fully documented so that potential users can decide whether or not the components meet their needs. The syntax and, ideally, the semantics of all component interfaces should be specified.

Figure 17.1 Component characteristics

proposals. Figure 17.1 shows what I consider to be the essential characteristics of a component as used in CBSE.

A useful way of thinking about a component is as a provider of one or more services. When a system needs a service, it calls on a component to provide that service without caring about where that component is executing or the programming language used to develop the component. For example, a component in a library system might provide a search service that allows users to search different library catalogs. A component that converts from one graphical format to another (e.g., TIFF to JPEG) provides a data conversion service, etc.

Viewing a component as a service provider emphasizes two critical characteristics of a reusable component:

1. The component is an independent executable entity that is defined by its interfaces. You don't need any knowledge of its source code to use it. It can either be referenced as an external service or included directly in a program.
2. The services offered by a component are made available through an interface and all interactions are through that interface. The component interface is expressed in terms of parameterized operations and its internal state is never exposed.

Component and objects

Components are often implemented in object-oriented languages and, in some cases, accessing the 'provides' interface of a component is done through method calls. However, components and object classes are not the same thing. Unlike object classes, components are independently deployable, do not define types, are language-independent, and are based on a standard component model.

http://www.SoftwareEngineering-9.com/Web/CBSE/objects.html

Components have two related interfaces, as shown in Figure 17.2. These interfaces reflect the services that the component provides and the services that the component requires to operate correctly:

- The 'provides' interface defines the services provided by the component. This interface, essentially, is the component API. It defines the methods that can be called by a user of the component. In a UML component diagram, the 'provides' interface for a component is indicated by a circle at the end of a line from the component icon.
- The 'requires' interface specifies what services must be provided by other components in the system if a component is to operate correctly. If these are not available, then the component will not work. This does not compromise the independence or deployability of a component because the 'requires' interface does not define how these services should be provided. In the UML, the symbol for a 'requires' interface is a semicircle at the end of a line from the component icon. Notice that 'provides' and 'requires' interface icons can fit together like a ball and socket.

To illustrate these interfaces, Figure 17.3 shows a model of a component that has been designed to collect and collate information from an array of sensors. It runs autonomously to collect data over a period of time and, on request, provides collated data to a calling component. The 'provides' interface includes methods to add, remove, start, stop, and test sensors. The report method returns the sensor data that has been collected, and the listAll method provides information about the attached sensors. Although I have not shown this here, these methods have associated parameters specifying the sensor identifiers, locations, and so on.

The 'requires' interface is used to connect the component to the sensors. It assumes that sensors have a data interface, accessed through sensorData, and a management

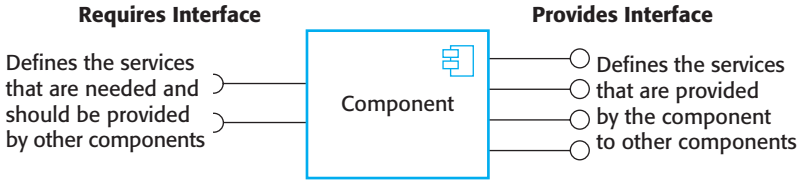

Figure 17.2 Component interfaces

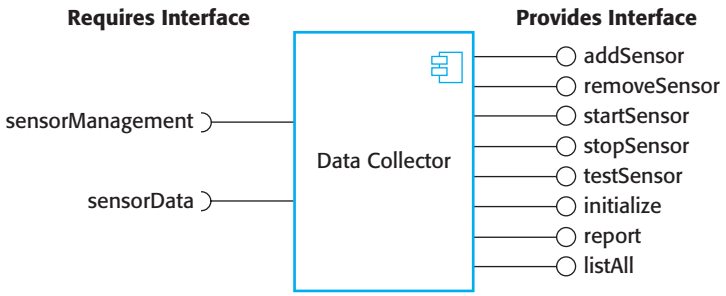

Figure 17.3 A model of a data collector component

interface, accessed through sensorManagement. This interface has been designed to connect to different types of sensor so it does not include specific sensor operations such as Test, provideReading, etc. Instead, the commands used by a specific type of sensor are embedded in a string, which is a parameter to the operations in the 'requires' interface. Adaptor components parse this string and translate the embedded commands into the specific control interface of each type of sensor. I discuss the use of adaptors later in this chapter, where I show how the data collector component is linked to a sensor (Figure 17.12).

A critical difference between a component as an external service and a component as a program element is that services are completely independent entities. They do not have a 'requires' interface. Different programs can use these services without the need to implement any additional support required by the service.

17.1.1 Component models

A component model is a definition of standards for component implementation, documentation, and deployment. These standards are for component developers to ensure that components can interoperate. They are also for providers of component execution infrastructures who provide middleware to support component operation. Many component models have been proposed, but the most important models are now the WebServices model, Sun's Enterprise Java Beans (EJB) model, and Microsoft's .NET model (Lau and Wang, 2007).

The basic elements of an ideal component model are discussed by Weinreich and Sametinger (2001). I summarize these model elements in Figure 17.4. This diagram shows that the elements of a component model define the component interfaces, the information that you need to use the component in a program, and how a component should be deployed:

1. *Interfaces* Components are defined by specifying their interfaces. The component model specifies how the interfaces should be defined and the elements, such as operation names, parameters, and exceptions, which should be included in the interface definition. The model should also specify the language used to define the component interfaces. For web services, this is WSDL, which I discuss in

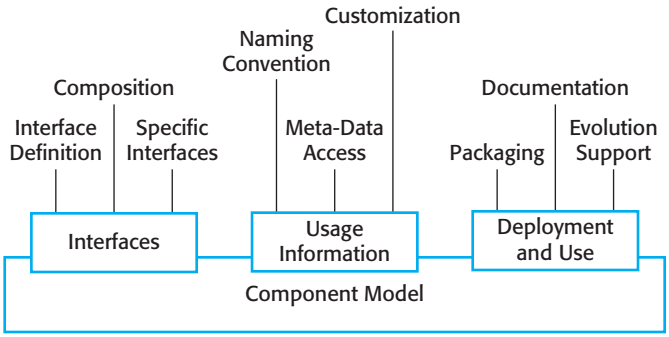

Figure 17.4 Basic elements of a component model

Chapter 19; EJB is Java-specific so Java is used as the interface definition language; in .NET, interfaces are defined using the Common Intermediate Language (CIL). Some component models require specific interfaces that must be defined by a component. These are used to compose the component with the component model infrastructure, which provides standardized services such as security and transaction management.

2. *Usage* In order for components to be distributed and accessed remotely, they need to have a unique name or handle associated with them. This has to be globally unique—for example, in EJB, a hierarchical name is generated with the root based on an Internet domain name. Services have a unique URI (Uniform Resource Identifier).

   Component meta-data is data about the component itself, such as information about its interfaces and attributes. The meta-data is important because it allows users of the component to find out what services are provided and required. Component model implementations normally include specific ways (such as the use of a reflection interface in Java) to access this component meta-data.

   Components are generic entities and, when deployed, they have to be configured to fit into an application system. For example, you could configure the Data collector component (Figure 17.2) by defining the maximum number of sensors in a sensor array. The component model may therefore specify how the binary components can be customized for a particular deployment environment.

3. *Deployment* The component model includes a specification of how components should be packaged for deployment as independent, executable entities. Because components are independent entities, they have to be packaged with all supporting software that is not provided by the component infrastructure, or is not defined in a 'requires' interface. Deployment information includes information about the contents of a package and its binary organization.

   Inevitably, as new requirements emerge, components will have to be changed or replaced. The component model may therefore include rules governing when and how component replacement is allowed. Finally, the component model may

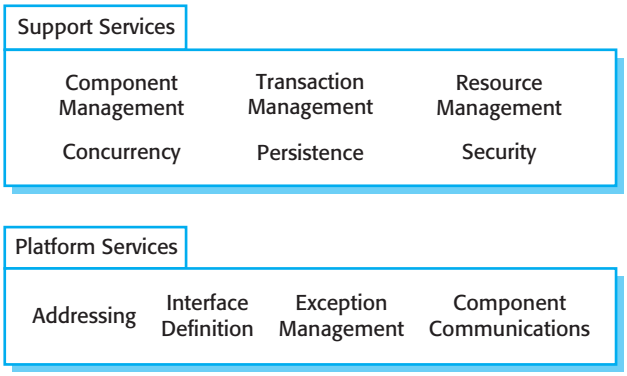

Figure 17.5 Middleware services defined in a component model

define the component documentation that should be produced. This is used to find the component and to decide whether it is appropriate.

For components that are implemented as program units rather than external services, the component model sets out the services to be provided by the middleware that supports the executing components. Weinreich and Sametinger (2001) use the analogy of an operating system to explain component models. An operating system provides a set of generic services that can be used by applications. A component model implementation provides comparable shared services for components. Figure 17.5 shows some of the services that may be provided by an implementation of a component model.

The services provided by a component model implementation fall into two categories:

1. Platform services, which enable components to communicate and interoperate in a distributed environment. These are the fundamental services that must be available in all component-based systems.
2. Support services, which are common services that are likely to be required by many different components. For example, many components require authentication to ensure that the user of component services is authorized. It makes sense to provide a standard set of middleware services for use by all components. This reduces the costs of component development and potential component incompatibilities can be avoided.

The middleware implements the component services and provides interfaces to these services. To make use of the services provided by a component model infrastructure, you can think of the components as being deployed in a 'container'. A container is an implementation of the support services plus a definition of the interfaces that a component must provide to integrate it with the container. Including the component in the container means that the component can access the support services and the container can access the component interfaces. When in use, the component interfaces themselves are not accessed directly by other components;

rather, they are accessed through a container interface that invokes code to access the interface of the embedded component.

Containers are large and complex and, when you deploy a component in a container, you get access to all middleware services. However, simple components may not need all of the facilities offered by the supporting middleware. The approach taken in web services to common service provision is therefore rather different. For web services, standards have been defined for common services such as transaction management and security and these standards have been implemented as program libraries. If you are implementing a service component, you only use the common services that you need.

17.2 CBSE processes

CBSE processes are software processes that support component-based software engineering. They take into account the possibilities of reuse and the different process activities involved in developing and using reusable components. Figure 17.6 (Kotonya, 2003) presents an overview of the processes in CBSE. At the highest level, there are two types of CBSE processes:

1. *Development for reuse* This process is concerned with developing components or services that will be reused in other applications. It usually involves generalizing existing components.
2. *Development with reuse* This is the process of developing new applications using existing components and services.

These processes have different objectives and therefore, include different activities. In the development for reuse process, the objective is to produce one or more reusable components. You know the components that you will be working with and you have access to their source code to generalize them. In development with reuse, you don't know what components are available, so you need to discover these components and design your system to make the most effective use of them. You may not have access to the component source code.

You can see from Figure 17.6 that the basic processes of CBSE with and for reuse have supporting processes that are concerned with component acquisition, component management, and component certification:

1. Component acquisition is the process of acquiring components for reuse or development into a reusable component. It may involve accessing locally developed components or services or finding these components from an external source.
2. Component management is concerned with managing a company's reusable components, ensuring that they are properly cataloged, stored, and made available for reuse.

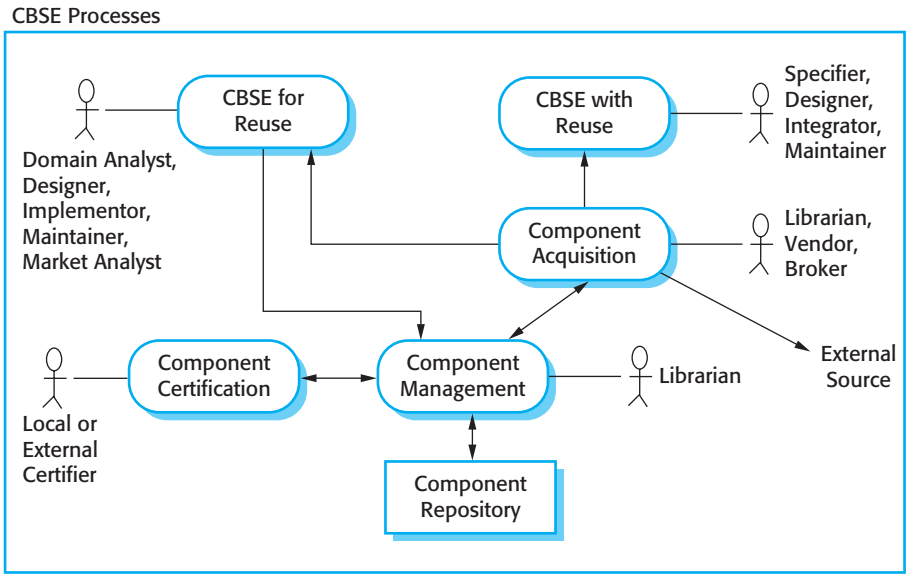

Figure 17.6 CBSE processes

3. Component certification is the process of checking a component and certifying that it meets its specification.

Components maintained by an organization may be stored in a component repository that includes both the components and information about their use.

17.2.1 CBSE for reuse

CBSE for reuse is the process of developing reusable components and making them available for reuse through a component management system. The vision of early supporters of CBSE (Szyperski, 2002) was that a thriving component marketplace would develop. There would be specialist component providers and component vendors who would organize the sale of components from different developers. Software developers would buy components to include in a system or pay for services as they were used. However, this vision has not been realized. There are relatively few component suppliers and buying components is uncommon. At the time of writing, the service market is also undeveloped although there are predictions that it will expand significantly over the next few years.

Consequently, CBSE for reuse is most likely to take place within an organization that has made a commitment to reuse-driven software engineering. They wish to exploit the software assets that have been developed in different parts of the company. However, these internally developed components are not usually reusable without change. They often include application-specific features and interfaces that are unlikely to be required in other programs where the component is reused.

To make components reusable, you have to adapt and extend the application-specific components to create more generic and therefore more reusable versions. Obviously, this adaptation has an associated cost. Thus you have to decide first, whether a component is likely to be reused and second, whether the cost savings from future reuse justify the costs of making the component reusable.

To answer the first of these questions, you have to decide whether or not the component implements one or more stable domain abstractions. Stable domain abstractions are fundamental elements of the application domain that change slowly. For example, in a banking system, domain abstractions might include accounts, account holders, and statements. In a hospital management system, domain abstractions might include patients, treatments, and nurses. These domain abstractions are sometimes called 'business objects'. If the component is an implementation of a commonly used domain abstraction or group of related business objects, it can probably be reused.

To answer the question about the cost effectiveness, you have to assess the costs of changes that are required to make the component reusable. These costs are the costs of component documentation, component validation, and making the component more generic. Changes that you may make to a component to make it more reusable include:

- removing application-specific methods;
- changing names to make them more general;
- adding methods to provide more complete functional coverage;
- making exception handling consistent for all methods;
- adding a 'configuration' interface to allow the component to be adapted to different situations of use;
- integrating required components to increase independence.

The problem of exception handling is a particularly difficult one. Components should not handle exceptions themselves, because each application will have its own requirements for exception handling. Rather, the component should define what exceptions can arise and should publish these as part of the interface. For example, a simple component implementing a stack data structure should detect and publish stack overflow and stack underflow exceptions. In practice, however, there are two problems with this:

1. Publishing all exceptions leads to bloated interfaces that are harder to understand. This may put off potential users of the component.
2. The operation of the component may depend on local exception handling, and changing this may have serious implications for the functionality of the component.

Mili et al. (2002) discuss ways of estimating the costs of making a component reusable and the returns from that investment. The benefits of reusing rather than redeveloping a component are not simply productivity gains. There are also quality gains, because a reused component should be more dependable, and time-to-market gains. These are the increased returns that accrue from deploying the software more quickly. Mili et al. present various formulas for estimating these gains, as does the COCOMO model discussed in Chapter 23 (Boehm, et al., 2000). However, the parameters of these formulas are difficult to estimate accurately, and the formulas must be adapted to local circumstances, making them difficult to use. I suspect that few software project managers use these models to estimate the return on investment from component reusability.

Obviously, whether or not a component is reusable depends on its application domain and functionality. As you add generality to a component, you increase its reusability. However, this normally means that the component has more operations and is more complex, which makes the component harder to understand and use.

There is, therefore, an inevitable trade-off between reusability and usability of a component. To make a component reusable you have to provide a set of generic interfaces with operations that cater to all of the ways in which the component could be used. Making the component usable means providing a simple, minimal interface that is easy to understand. Reusability adds complexity and hence reduces component understandability. It is therefore more difficult to decide when and how to reuse that component. When designing a reusable component, you must, therefore, find a compromise between generality and understandability.

A potential source of components is existing legacy systems. As I discussed in Chapter 9, these are systems that fulfill an important business function but are written using obsolete software technologies. Because of this, it may be difficult to use them with new systems. However, if you convert these old systems to components, their functionality can be reused in new applications.

Of course, these legacy systems do not normally have clearly defined ‘requires’ and ‘provides’ interfaces. To make these components reusable, you have to create a wrapper that defines the component interfaces. The wrapper hides the complexity of the underlying code and provides an interface for external components to access services that are provided. Although this wrapper is a fairly complex piece of software, the cost of wrapper development is often much less than the cost of reimplementing the legacy system. I discuss this approach in more detail in Chapter 19, where I explain how the features in a legacy system can be accessed through services.

Once you have developed and tested a reusable component or service, this then has to be managed for future reuse. Management involves deciding how to classify the component so that it can be discovered, making the component available either in a repository or as a service, maintaining information about the use of the component and keeping track of different component versions. If the component is open source, you may make it available in a public repository such as Sourceforge. If it is intended for use in a company, then you may use an internal repository system.

A company with a reuse program may carry out some form of component certification before the component is made available for reuse. Certification means that

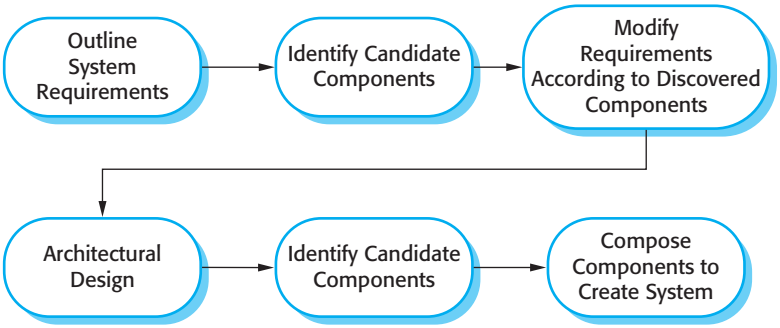

Figure 17.7 CBSE with reuse

someone apart from the developer checks the quality of the component. They test the component and certify that it has reached an acceptable quality standard, before it is made available for reuse. However, this can be an expensive process and many companies simply leave testing and quality checking to the component developers.

17.2.2 CBSE with reuse

The successful reuse of components requires a development process tailored to CBSE. The CBSE with reuse process has to include activities that find and integrate reusable components. The structure of such a process was discussed in Chapter 2 and Figure 17.7 shows the principal activities within that process. Some of the activities within this process, such as the initial discovery of user requirements, are carried out in the same way as in other software processes. However, the essential differences between CBSE with reuse and software processes for original software development are:

1. The user requirements are initially developed in outline rather than in detail, and stakeholders are encouraged to be as flexible as possible in defining their requirements. Requirements that are too specific limit the number of components that could meet these requirements. However, unlike incremental development, you need a complete set of requirements so that you can identify as many components as possible for reuse.
2. Requirements are refined and modified early in the process depending on the components available. If the user requirements cannot be satisfied from available components, you should discuss the related requirements that can be supported. Users may be willing to change their minds if this means cheaper or quicker system delivery.
3. There is a further component search and design refinement activity after the system architecture has been designed. Some apparently usable components may turn out to be unsuitable or do not work properly with other chosen components. Although not shown in Figure 17.7, this implies that further requirements changes may be necessary.

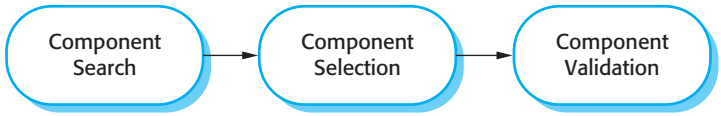

Figure 17.8 The component identification process

4. Development is a composition process where the discovered components are integrated. This involves integrating the components with the component model infrastructure and, often, developing adaptors that reconcile the interfaces of incompatible components. Of course, additional functionality may also be required over and above that provided by reused components.

The architectural design stage is particularly important. Jacobson et al. (1997) found that defining a robust architecture is critical for successful reuse. During the architectural design activity, you may choose a component model and implementation platform. However, many companies have a standard development platform (e.g., .NET) so the component model is pre-determined. As I discussed in Chapter 6, you also establish the high-level organization of the system at this stage and make decisions about system distribution and control.

An activity that is unique to the CBSE process is identifying candidate components or services for reuse. This involves a number of subactivities, as shown in Figure 17.8. Initially, your focus should be on search and selection. You need to convince yourself that there are components available to meet your requirements. Obviously, you should do some initial checking that the component is suitable but detailed testing may not be required. In the later stage, after the system architecture has been designed, you should spend more time on component validation. You need to be confident that the identified components are really suited to your application; if not, then you have to repeat the search and selection processes.

The first step in identifying components is to look for components that are available locally or from trusted suppliers. As I said in the previous section, there are relatively few component vendors so you are therefore most likely to be looking for components that have been developed in your own company. Software development companies can build their own database of reusable components without the risks inherent in using components from external suppliers. Alternatively, you may decide to search code libraries available on the Web, such as Sourceforge or Google Code, to see if source code for the component that you need is available. If you are looking for services, then there are a number of specialized web search engines available that can discover public web services.

Once the component search process has identified possible components, you have to select candidate components for assessment. In some cases, this will be a straightforward task. Components on the list will directly implement the user requirements and there will not be competing components that match these requirements. In other cases, however, the selection process is much more complex. There will not be a clear mapping of requirements onto components and you may find that several components have to be integrated to meet a specific requirement or group of requirements.

The Ariane 5 launcher Failure

While developing the Ariane 5 space launcher, the designers decided to reuse the inertial reference software that had performed successfully in the Ariane 4 launcher. The inertial reference software maintains the stability of the rocket. They decided to reuse this without change (as you would do with components), although it included additional functionality that was not required in Ariane 5.

In the first launch of Ariane 5, the inertial navigation software failed and the rocket could not be controlled. Ground controllers instructed the launcher to self-destruct and the rocket and its payload were destroyed. The cause of the problem was an unhandled exception when a conversion of a fixed-point number to an integer resulted in a numeric overflow. This caused the run-time system to shut down the inertial reference system and launcher stability could not be maintained. The fault had never occurred in Ariane 4 because it had less powerful engines and the value that was converted could not be large enough for the conversion to overflow.

The fault occurred in code that was not required for Ariane 5. The validation tests for the reused software were based on Ariane 5 requirements. Because there were no requirements for the function that failed, no tests were developed. Consequently, the problem with the software was never discovered during launch simulation tests.

Figure 17.9 An example of validation failure with reused software

You, therefore, have to decide which component compositions provide the best coverage of the requirements.

Once you have selected components for possible inclusion in a system, you should then validate them to check that they behave as advertised. The extent of the validation required depends on the source of the components. If you are using a component that has been developed by a known and trusted source, you may decide that component testing is unnecessary. You simply test the component when it is integrated with other components. On the other hand, if you are using a component from an unknown source, you should always check and test that component before including it in your system.

Component validation involves developing a set of test cases for a component (or, possibly, extending test cases supplied with that component) and developing a test harness to run component tests. The major problem with component validation is that the component specification may not be sufficiently detailed to allow you to develop a complete set of component tests. Components are usually specified informally, with the only formal documentation being their interface specification. This may not include enough information for you to develop a complete set of tests that would convince you that the component's advertised interface is what you require.

As well as testing that a component for reuse does what you require, you may also have to check that the component does not include any malicious code or functionality that you don't need. Professional developers rarely use components from untrusted sources, especially if these sources do not provide source code. Therefore, the malicious code problem does not usually arise. However, components may often contain functionality that you don't need and you have to check that this functionality will not interfere with your use of the component.

The problem with unnecessary functionality is that it may be activated by the component itself. This can slow down the component, cause it to produce surprising results or, in some cases, cause serious system failures. Figure 17.9 summarizes a situation where unnecessary functionality in a reused system caused a catastrophic software failure.

The problem in the Ariane 5 launcher arose because the assumptions made about the software for Ariane 4 were invalid for Ariane 5. This is a general problem with reusable components. They are originally implemented for an application environment and, naturally, embed assumptions about that environment. These assumptions are rarely documented so, when the component is reused, it is impossible to derive tests to check if the assumptions are still valid. If you are reusing a component in a different environment, you may not discover the embedded environmental assumptions until you use the component in an operational system.

17.3 Component composition

Component composition is the process of integrating components with each other, and with specially written 'glue code' to create a system or another component. There are several different ways in which you can compose components, as shown in Figure 17.10. From left to right these diagrams illustrate sequential composition, hierarchical composition and additive composition. In the discussion below, I assume that you are composing two components (A and B) to create a new component:

1. Sequential composition is situation (a) in Figure 17.10. You create a new component from 2 existing components by calling the existing components in sequence. You can think of the composition as a composition of the 'provides interfaces'. That is, the services offered by component A are called and the results returned by A are then used in the call to the services offered by component B. The components do not call each other in sequential composition. Some extra glue code is required to call the component services in the right order and to ensure that the results delivered by component A are compatible with the inputs expected by component B. The 'provides' interface of the composition depends on the combined functionality of A and B but will not normally be a composition of their 'provides interfaces'. This type of composition may be used with components that are program elements or components that are services.
2. Hierarchical composition is situation (b) in Figure 17.10. This type of composition occurs when one component calls directly on the services provided by another component. The called component provides the services that are required by the calling component. Therefore, the 'provides' interface of the called component must be compatible with the 'requires' interface of the calling component. Component A calls on component B directly and, if their interfaces match, there may be no need for additional code. However, if there is a mismatch between the 'requires' interface of A and the 'provides' interface of B, then some conversion code may be required. As services do not have a 'requires' interface, this mode of composition is not used when components are implemented as web services.
3. Additive composition corresponds to situation (c) in Figure 17.10. This occurs when two or more components are put together (added) to create a new component,

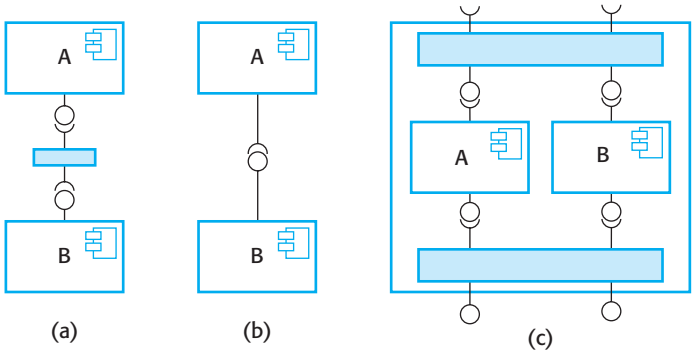

Figure 17.10 Types of component composition

which combines their functionality. The 'provides' interface and 'requires' interface of the new component is a combination of the corresponding interfaces in components A and B. The components are called separately through the external interface of the composed component. A and B are not dependent and do not call each other. This type of composition may be used with components that are program units or components that are services.

You might use all the forms of component composition when creating a system. In all cases, you may have to write 'glue code' that links the components. For example, for sequential composition, the output of component A typically becomes the input to component B. You need intermediate statements that call component A, collect the result, and then call component B with that result as a parameter. When one component calls another, you may need to introduce an intermediate component that ensures that the 'provides' interface and the 'requires' interface are compatible.

When you write new components especially for composition, you should design the interfaces of these components so that they are compatible with other components in the system. You can therefore easily compose these components into a single unit. However, when components are developed independently for reuse, you will often be faced with interface incompatibilities. This means that the interfaces of the components that you wish to compose are not the same. Three types of incompatibility can occur:

1. *Parameter incompatibility* The operations on each side of the interface have the same name but their parameter types or the number of parameters are different.
2. *Operation incompatibility* The names of the operations in the 'provides' and 'requires' interfaces are different.
3. *Operation incompleteness* The 'provides' interface of a component is a subset of the 'requires' interface of another component or vice versa.

In all cases, you tackle the problem of incompatibility by writing an adaptor that reconciles the interfaces of the two components being reused. An adaptor component converts one interface to another. The precise form of the adaptor depends on the type

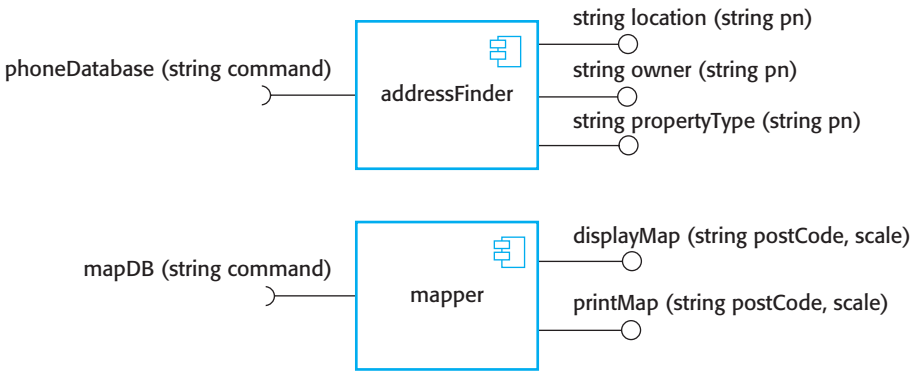

Figure 17.11 Components with incompatible interfaces

of composition. Sometimes, as in the next example, the adaptor takes a result from one component and converts it into a form where it can be used as an input to another. In other cases, the adaptor may be called by component A as a proxy for component B. This situation occurs if A wishes to call B but the details of the 'requires' interface of A do not match the details of the 'provides' interface of B. The adaptor reconciles these differences by converting its input parameters from A into the required input parameters for B. It then calls B to deliver the services required by A.

To illustrate adaptors, consider the two components shown in Figure 17.11, whose interfaces are incompatible. These might be part of a system used by the emergency services. When the emergency operator takes a call, the phone number is input to the addressFinder component to locate the address. Then, using the mapper component, the operator prints a map to be sent to the vehicle dispatched to the emergency. In fact, the components would have more complex interfaces than those shown here, but the simplified version illustrates the concept of an adaptor.

The first component, addressFinder, finds the address that matches a phone number. It can also return the owner of the property associated with the phone number and the type of property. The mapper component takes a post code (in the United States, a standard ZIP code with the additional four digits identifying property location) and displays or prints a street map of the area around that code at a specified scale.

These components are composable in principle because the property location includes the post or ZIP code. However, you have to write an adaptor component called postCodeStripper that takes the location data from addressFinder and strips out the post code. This post code is then used as an input to mapper and the street map is displayed at a scale of 1:10,000. The following code, which is an example of sequential composition, illustrates the sequence of calls that is required to implement this:

```
address = addressFinder.location (phonenumber) ;
postCode = postCodeStripper.getPostCode (address) ;
mapper.displayMap(postCode, 10000) ;
```

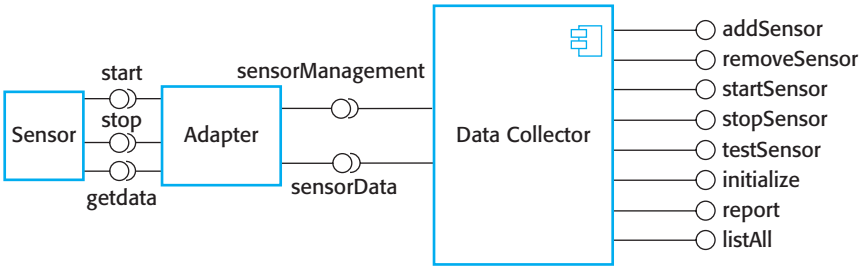

Figure 17.12 An adaptor linking a data collector and a sensor

Another case in which an adaptor component may be used is in hierarchical composition, where one component wishes to make use of another but there is an incompatibility between the 'provides' interface and 'requires' interface of the components in the composition. I have illustrated the use of an adaptor in Figure 17.12 where an adaptor is used to link a data collector and a sensor component. These could be used in the implementation of a wilderness weather station system, as discussed in Chapter 7.

The sensor and data collector components are composed using an adapter that reconciles the 'requires' interface of the data collection component with the 'provides' interface of the sensor component. The data collector component has been designed with a generic 'requires' interface that supports sensor data collection and sensor management. For each of these operations, the parameter is a text string representing the specific sensor commands. For example, to issue a collect command, you would say sensorData("collect"). As I have shown in Figure 17.12, the sensor itself has separate operations such as start, stop, and getdata.

The adaptor parses the input string, identifies the command (e.g., collect) and then calls Sensor.getdata to collect the sensor value. It then returns the result (as a character string) to the data collector component. This interface style means that the data collector can interact with different types of sensor. A separate adaptor, which converts the sensor commands from Data collector to the actual sensor interface, is implemented for each type of sensor.

The above discussion of component composition assumes you can tell from the component documentation whether or not interfaces are compatible. Of course, the interface definition includes the operation name and parameter types, so you can make some assessment of the compatibility from this. However, you depend on the component documentation to decide whether the interfaces are semantically compatible.

To illustrate this problem, consider the composition shown in Figure 17.13. These components are used to implement a system that downloads images from a digital camera and stores them in a photograph library. The system user can provide additional information to describe and catalog the photograph. To avoid clutter,

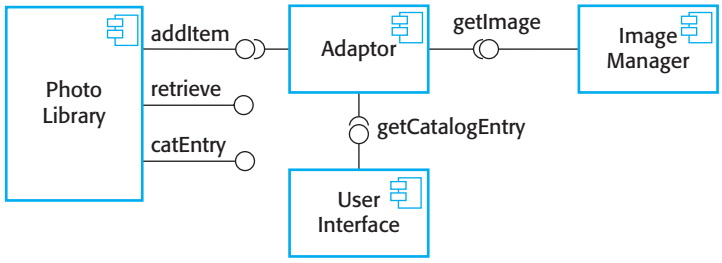

Figure 17.13 Photo library composition

I have not shown all interface methods here. Rather, I simply show the methods that are needed to illustrate the component documentation problem. The methods in the interface of Photo Library are:

```
public void addItem (Identifier pid ; Photograph p; CatalogEntry
photodesc) ;
public Photograph retrieve (Identifier pid) ;
public CatalogEntry catEntry (Identifier pid);
```

Assume that the documentation for the addItem method in Photo Library is:

This method adds a photograph to the library and associates the photograph identifier and catalogue descriptor with the photograph.

This description appears to explain what the component does, but consider the following questions:

- What happens if the photograph identifier is already associated with a photograph in the library?
- Is the photograph descriptor associated with the catalog entry as well as the photograph? That is, if you delete the photograph, do you also delete the catalog information?

There is not enough information in the informal description of addItem to answer these questions. Of course, it is possible to add more information to the natural language description of the method, but in general, the best way to resolve ambiguities is to use a formal language to describe the interface. The specification shown in Figure 17.14 is part of the description of the interface of Photo Library that adds information to the informal description.

The specification in Figure 17.14 uses pre- and post-conditions that are defined in a notation based on the object constraint language (OCL), which is part of the UML (Warmer and Kleppe, 2003). OCL is designed to describe constraints in UML object models; it allows you to express predicates that must always be true, that must be

```
— The context keyword names the component to which the conditions apply

context addItem

— The preconditions specify what must be true before execution of addItem
pre:     PhotoLibrary.libSize() > 0
         PhotoLibrary.retrieve(pid) = null

— The postconditions specify what is true after execution
post:    libSize () = libSize()@pre + 1
         PhotoLibrary.retrieve(pid) = p
         PhotoLibrary.catEntry(pid) = photodesc

context delete

pre:     PhotoLibrary.retrieve(pid) <>null ;

post:    PhotoLibrary.retrieve(pid) = null
         PhotoLibrary.catEntry(pid) = PhotoLibrary.catEntry(pid)@pre
         PhotoLibrary.libSize() = libSize()@pre—1
```

Figure 17.14 The OCL description of the Photo Library interface

true before a method has executed; and that must be true after a method has executed. These are invariants, pre-conditions, and post-conditions. To access the value of a variable before an operation, you add @pre after its name. Therefore, using age as an example:

```
age = age@pre + 1
```

This statement means that the value of age after an operation is one more than it was before that operation.

OCL-based approaches are increasingly used to add semantic information to UML models, and OCL descriptions may be used to drive code generators in model-driven engineering. The general approach has been derived from Meyer's Design by Contract approach (Meyer, 1992), in which the interfaces and obligations of communicating objects are formally specified and enforced by the run-time system. Meyer suggests that using Design by Contract is essential if we are to develop trusted components (Meyer, 2003).

Figure 17.14 includes a specification for the addItem and delete methods in Photo Library. The method being specified is indicated by the keyword context and the pre- and post-conditions by the keywords pre and post. The pre-conditions for addItem state that:

1. There must not be a photograph in the library with the same identifier as the photograph to be entered.
2. The library must exist—assume that creating a library adds a single item to it so that the size of a library is always greater than zero.

3. The post-conditions for addItem state that:

   The size of the library has increased by 1 (so only a single entry has been made).

   If you retrieve using the same identifier, then you get back the photograph that you added.

   If you look up the catalogue using that identifier, you get back the catalogue entry that you made.

The specification of delete provides further information. The pre-condition states that to delete an item, it must be in the library and, after deletion, the photo can no longer be retrieved and the size of the library is reduced by 1. However, delete does not delete the catalogue entry—you can still retrieve it after the photo has been deleted. The reason for this is that you may wish to maintain information in the catalog about why a photo was deleted, its new location, and so on.

When you create a system by composing components, you may find that there are potential conflicts between functional and non-functional requirements, the need to deliver a system as quickly as possible and the need to create a system that can evolve as requirements change. The decisions where you may have to take trade-offs into account are:

1. What composition of components is most effective for delivering the functional requirements for the system?
2. What composition of the components will make it easier to adapt the composite component when its requirements change?
3. What will be the emergent properties of the composed system? These are properties such as performance and dependability. You can only assess these once the complete system is implemented.

Unfortunately, there are many situations where the solutions to the composition problems may conflict. For example, consider a situation such as that illustrated in Figure 17.15, where a system can be created through two alternative compositions. The system is a data collection and reporting system where data is collected from different sources, stored in a database and then different reports summarizing that data are produced.

Here, there is a potential conflict between adaptability and performance. Composition (a) is more adaptable but composition (b) is perhaps faster and more reliable. The advantages of composition (a) are that reporting and data management are separate, so there is more flexibility for future change. The data management system could be replaced and, if reports are required that the current reporting component cannot produce, that component can also be replaced without having to change the data management component.

In composition (b), a database component with built-in reporting facilities (e.g., Microsoft Access) is used. The key advantage of composition (b) is that there are fewer components, so this will be a faster implementation because there are no component

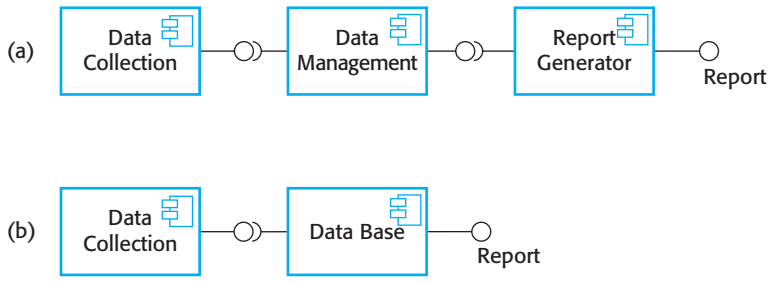

Figure 17.15 Data collection and report generation components

communication overheads. Furthermore, data integrity rules that apply to the database will also apply to reports. These reports will not be able to combine data in incorrect ways. In composition (a), there are no such constraints so errors in reports could occur.

In general, a good composition principle to follow is the principle of separation of concerns. That is, you should try to design your system in such a way that each component has a clearly defined role and that, ideally, these roles should not overlap. However, it may be cheaper to buy one multi-functional component rather than two or three separate components. Furthermore, there may be dependability or performance penalties when multiple components are used.

KEY POINTS

- Component-based software engineering is a reuse-based approach to defining, implementing, and composing loosely coupled independent components into systems.
- A component is a software unit whose functionality and dependencies are completely defined by a set of public interfaces. Components can be composed with other components without knowledge of their implementation and can be deployed as an executable unit.
- Components may be implemented as program units that are included in a system or as external services that are referenced from within a system.
- A component model defines a set of standards for components, including interface standards, usage standards, and deployment standards. The implementation of the component model provides a set of common services that may be used by all components.
- During the CBSE process, you have to interleave the processes of requirements engineering and system design. You have to trade off desirable requirements against the services that are available from existing reusable components.
- Component composition is the process of 'wiring' components together to create a system. Types of composition include sequential composition, hierarchical composition, and additive composition.

- When composing reusable components that have not been written for your application, you may need to write adaptors or 'glue code' to reconcile the different component interfaces.
- When choosing compositions, you have to consider the required functionality of the system, the non-functional requirements and the ease with which one component can be replaced when the system is changed.

FURTHER READING

Component-based Software Engineering: Putting the Pieces Together. This book is a collection of papers from various authors on different aspects of CBSE. Like all collections, it is rather mixed but it has better coverage of general issues of software engineering with components than Szyperski's book. (G. T. Heineman and W. T. Councill, Addison-Wesley, 2001.)

Component Software: Beyond Object-Oriented Programming, 2nd ed. This updated edition of the first book on CBSE covers technical and nontechnical issues in CBSE. It has more detail on specific technologies than Heineman and Councill's book and includes a thorough discussion of market issues. (C. Szyperski, Addison-Wesley, 2002.)

'Specification, Implementation and Deployment of Components'. A good introduction to the fundamentals of CBSE. The same issue of the *CACM* includes articles on components and component-based development. (I. Crnkovic, B. Hnich, T. Jonsson and Z. Kiziltan, *Comm. ACM,* **45** (10), October 2002.) http://dx.doi.org/10.1145/570907.570928.

'Software Component Models'. This is a comprehensive discussion of commercial and research component models that classifies these models and explains the differences between them. (K-K. Lau and Z. Wang, *IEEE Transactions on Software Engineering,* **33** (10), October 2007.) http://dx.doi.org/10.1109/TSE.2007.70726.

EXERCISES

17.1. Why is it important that all component interactions are defined through 'requires' and 'provides' interfaces?

17.2. The principle of component independence means that it ought to be possible to replace one component with another that is implemented in a completely different way. Using an example, explain how such component replacement could have undesired consequences and may lead to system failure.

17.3. What are the fundamental differences between components as program elements and components as services?

17.4. Why is it important that components should be based on a standard component model?

17.5. Using an example of a component that implements an abstract data type such as a stack or a list, show why it is usually necessary to extend and adapt components for reuse.

17.6. Explain why it is difficult to validate a reusable component without the component source code. In what ways would a formal component specification simplify the problems of validation?

17.7. Design the 'provides' interface and the 'requires' interface of a reusable component that may be used to represent a patient in the MHC-PMS.

17.8. Using examples, illustrate the different types of adaptor needed to support sequential composition, hierarchical composition, and additive composition.

17.9. Design the interfaces of components that might be used in a system for an emergency control room. You should design interfaces for a call-logging component that records calls made, and a vehicle discovery component that, given a post code (zip code) and an incident type, finds the nearest suitable vehicle to be despatched to the incident.

17.10. It has been suggested that an independent certification authority should be established. Vendors would submit their components to this authority, which would validate that the component was trustworthy. What would be the advantages and disadvantages of such a certification authority?

REFERENCES

Boehm, B. W., Abts, C., Brown, A. W., Chulani, S., Clark, B. K., Horowitz, E., Madachy, R., Reifer, D. and Steece, B. (2000). *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ.: Prentice Hall.

Councill, W. T. and Heineman, G. T. (2001). 'Definition of a Software Component and its Elements'. In *Component-based Software Engineering*. Heineman, G. T. and Councill, W. T. (ed.). Boston: Addison-Wesley, 5–20.

Jacobson, I., Griss, M. and Jonsson, P. (1997). *Software Reuse*. Reading, Mass.: Addison-Wesley.

Kotonya, G. (2003). 'The CBSE Process: Issues and Future Visions'. *Proc. 2nd CBSEnet workshop,* Budapest, Hungary.

Lau, K.-K. and Wang, Z. (2007). 'Software Component Models'. *IEEE Trans. on Software Eng.*, **33** (10), 709–24.

Meyer, B. (1992). 'Design by Contract'. *IEEE Computer,* **25** (10), 40–51.

Meyer, B. (2003). 'The Grand Challenge of Trusted Components'. *ICSE 25: Int. Conf. on Software Eng.*, Portland, Oregon: IEEE Press.