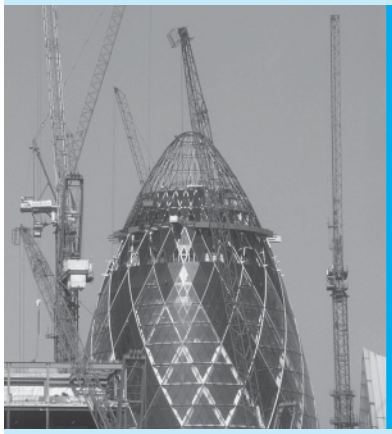


PART 3 Advanced Software Engineering

This part of the book covers more advanced software engineering topics. I assume in these chapters that readers understand the basics of the discipline, covered in Chapters 1–9.

Chapters 15–18 focus on the dominant development paradigm for web-based information systems and enterprise systems—software reuse. Chapter 15 introduces the topic and explains the different types of reuse that are possible. I then cover the most common approach to reuse, which is the reuse of application systems. These are configured and adapted to the specific needs of each business.

Chapter 16 is concerned with the reuse of software components rather than entire software systems. In this chapter, I explain what is meant by a component and why standard component models are needed for effective component reuse. I also discuss the general process of component-based software engineering and the problems of component composition.

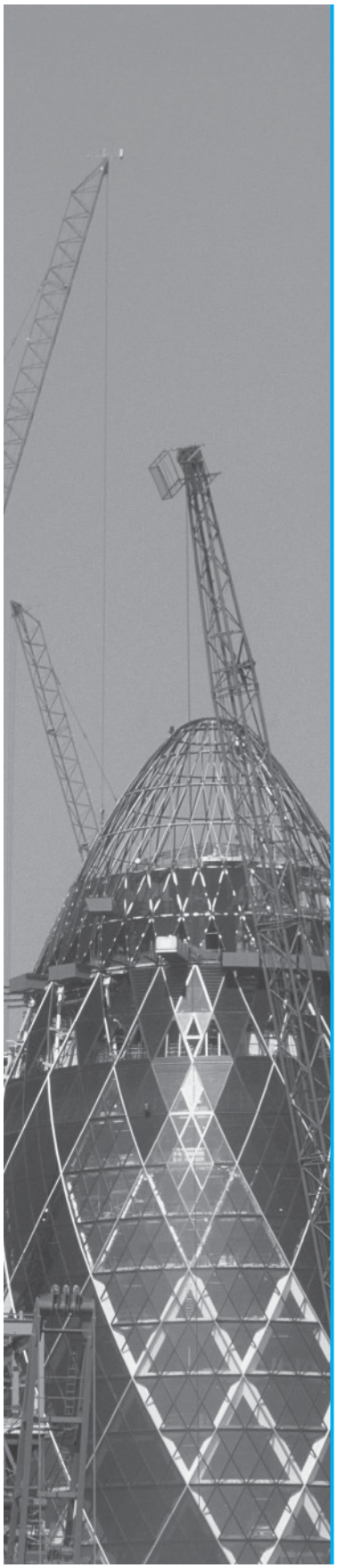
The majority of large systems are now distributed systems and Chapter 17 covers issues and problems of building distributed systems. I introduce the client-server approach as a fundamental paradigm of distributed systems engineering, and explain ways of implementing this architectural style. The final section explains software as a service–the delivery of software functionality over the Internet, which has changed the market for software products.

Chapter 18 introduces the related topic of service-oriented architectures, which link the notions of distribution and reuse. Services are reusable software components whose functionality can be accessed over the Internet. I discuss two widely-used approaches to service development namely SOAP-based and RESTful services. I explain what is involved in creating services (service engineering) and composing services to create new software systems.

The focus of Chapters 19–21 is systems engineering. In Chapter 19, I introduce the topic and explain why it is important that software engineers should understand systems engineering. I discuss the systems engineering life cycle and the importance of procurement in that life-cycle.

Chapter 20 covers systems of systems (SoS). The large systems that we will build in the 21st century will not be developed from scratch but will be created by integrating existing complex systems. I explain why an understanding of complexity is important in SoS development and discuss architectural patterns for complex systems of systems.

Most software systems are not apps or business systems but are embedded real-time systems. Chapter 21 covers this important topic. I introduce the idea of a real-time embedded system and describe architectural patterns that are used in embedded systems design. I then explain the process of timing analysis and conclude the chapter with a discussion of real-time operating systems.



15 Software reuse

Objectives

The objectives of this chapter are to introduce software reuse and to describe approaches to system development based on large-scale software reuse. When you have read this chapter, you will:

- understand the benefits and problems of reusing software when developing new systems;
- understand the concept of an application framework as a set of reusable objects and how frameworks can be used in application development;
- have been introduced to software product lines, which are made up of a common core architecture and reusable components that are configured for each version of the product;
- have learned how systems can be developed by configuring and composing off-the-shelf application software systems.

Contents

15.1 The reuse landscape
15.2 Application frameworks
15.3 Software product lines
15.4 Application system reuse

Reuse-based software engineering is a software engineering strategy where the development process is geared to reusing existing software. Until around 2000, systematic software reuse was uncommon, but it is now used extensively in the development of new business systems. The move to reuse-based development has been in response to demands for lower software production and maintenance costs, faster delivery of systems, and increased software quality. Companies see their software as a valuable asset. They are promoting reuse of existing systems to increase their return on software investments.

Reusable software of different kinds is now widely available. The open-source movement has meant that there is a huge code base that can be reused. This may be in the form of program libraries or entire applications. Many domain-specific application systems, such as ERP systems, are available that can be tailored and adapted to customer requirements. Some large companies provide a range of reusable components for their customers. Standards, such as web service standards, have made it easier to develop software services and reuse them across a range of applications.

Reuse-based software engineering is an approach to development that tries to maximize the reuse of existing software. The software units that are reused may be of radically different sizes. For example:

1. *System reuse* Complete systems, which may be made up of a number of application programs, may be reused as part of a system of systems (Chapter 20).
2. *Application reuse* An application may be reused by incorporating it without change into other systems or by configuring the application for different customers. Alternatively, application families or software product lines that have a common architecture, but that are adapted to individual customer requirements, may be used to develop a new system.
3. *Component reuse* Components of an application, ranging in size from subsystems to single objects, may be reused. For example, a pattern-matching system developed as part of a text-processing system may be reused in a database management system. Components may be hosted on the cloud or on private servers and may be accessible through an application programming interface (API) as services.
4. *Object and function reuse* Software components that implement a single function, such as a mathematical function, or an object class may be reused. This form of reuse, designed around standard libraries, has been common for the past 40 years. Many libraries of functions and classes are freely available. You reuse the classes and functions in these libraries by linking them with newly developed application code. In areas such as mathematical algorithms and graphics, where specialized, expensive expertise is needed to develop efficient objects and functions, reuse is particularly cost-effective.

All software systems and components that include generic functionality are potentially reusable. However, these systems or components are sometimes so

Benefit	Explanation
Accelerated development	Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time may be reduced.
Effective use of specialists	Instead of doing the same work over and over again, application specialists can develop reusable software that encapsulates their knowledge.
Increased dependability	Reused software, which has been tried and tested in working systems, should be more dependable than new software. Its design and implementation faults should have been found and fixed.
Lower development costs	Development costs are proportional to the size of the software being developed. Reusing software means that fewer lines of code have to be written.
Reduced process risk	The cost of existing software is already known, while the costs of development are always a matter of judgment. This is an important factor for project management because it reduces the margin of error in project cost estimation. This is especially true when large software components such as subsystems are reused.
Standards compliance	Some standards, such as user interface standards, can be implemented as a set of reusable components. For example, if menus in a user interface are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability because users make fewer mistakes when presented with a familiar interface.

Figure 15.1 Benefits of software reuse

specific that it is very expensive to modify them for a new situation. Rather than reuse the code, however, you can reuse the ideas that are the basis of the software. This is called concept reuse.

In concept reuse you do not reuse a software component; rather, you reuse an idea, a way of working, or an algorithm. The concept that you reuse is represented in an abstract notation, such as a system model, which does not include implementation detail. It can, therefore, be configured and adapted for a range of situations. Concept reuse is embodied in approaches such as design patterns (Chapter 7), configurable system products, and program generators. When concepts are reused, the reuse process must include an activity where the abstract concepts are instantiated to create executable components.

An obvious advantage of software reuse is that overall development costs are lower. Fewer software components need to be specified, designed, implemented, and validated. However, cost reduction is only one benefit of software reuse. I have listed other advantages of reusing software in Figure 15.1.

However, there are costs and difficulties associated with reuse (Figure 15.2). There is a significant cost associated with understanding whether or not a component is suitable for reuse in a particular situation, and in testing that component to ensure its dependability. These additional costs mean that the savings in development costs may not be less than anticipated. However, the other benefits of reuse still apply.

Problem	Explanation
Creating, maintaining, and using a component library	Populating a reusable component library and ensuring the software developers can use this library can be expensive. Development processes have to be adapted to ensure that the library is used.
Finding, understanding, and adapting reusable components	Software components have to be discovered in a library, understood, and sometimes adapted to work in a new environment. Engineers must be reasonably confident of finding a component in the library before they include a component search as part of their normal development process.
Increased maintenance costs	If the source code of a reused software system or component is not available, then maintenance costs may be higher because the reused elements of the system may become incompatible with changes made to the system.
Lack of tool support	Some software tools do not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. The software process assumed by these tools may not take reuse into account. This is more likely to be the case for tools that support embedded systems engineering than for object-oriented development tools.
"Not-invented-here" syndrome	Some software engineers prefer to rewrite components because they believe they can improve on them. This is partly to do with trust and partly to do with the fact that writing original software is seen as more challenging than reusing other people's software.

Figure 15.2 Problems with software reuse

As I discussed in Chapter 2, software development processes have to be adapted to take reuse into account. In particular, there has to be a requirements refinement stage where the requirements for the system are modified to reflect the reusable software that is available. The design and implementation stages of the system may also include explicit activities to look for and evaluate candidate components for reuse.

15.1 The reuse landscape

Over the past 20 years, many techniques have been developed to support software reuse. These techniques exploit the facts that systems in the same application domain are similar and have potential for reuse, that reuse is possible at different levels from simple functions to complete applications, and that standards for reusable components facilitate reuse. Figure 15.3 shows the "reuse landscape"—different ways of implementing software reuse. Each of these approaches to reuse is briefly described in Figure 15.4.

Given this array of techniques for reuse, the key question is "which is the most appropriate technique to use in a particular situation?" Obviously, the answer to this question depends on the requirements for the system being developed, the technology

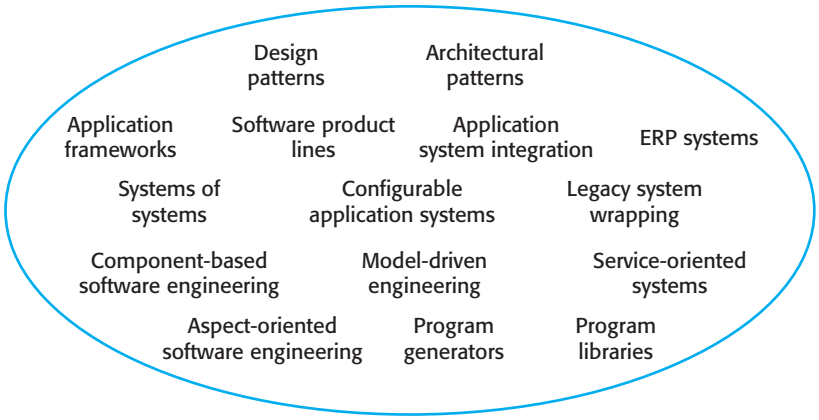


Figure 15.3 The reuse landscape

and reusable assets available, and the expertise of the development team. Key factors that you should consider when planning reuse are:

1. *The development schedule for the software* If the software has to be developed quickly, you should try to reuse complete systems rather than individual components. Although the fit to requirements may be imperfect, this approach minimizes the amount of development required.

2. *The expected software lifetime* If you are developing a long-lifetime system, you should focus on the maintainability of the system. You should not just think about the immediate benefits of reuse but also of the long-term implications.

   Over its lifetime, you will have to adapt the system to new requirements, which will mean making changes to parts of the system. If you do not have access to the source code of the reusable components, you may prefer to avoid off-the-shelf components and systems from external suppliers. These suppliers may not be able to continue support for the reused software. You may decide that it is safer to reuse open-source systems and components (Chapter 7) as this means you can access and keep copies of the source code.

3. *The background, skills and experience of the development team* All reuse technologies are fairly complex, and you need quite a lot of time to understand and use them effectively. Therefore, you should focus your reuse effort in areas where your development team has expertise.

4. *The criticality of the software and its non-functional requirements* For a critical system that has to be certified by an external regulator you may have to create a safety or security case for the system (discussed in Chapter 12). This is difficult if you don't have access to the source code of the software. If your software has stringent performance requirements, it may be impossible to use strategies such as model-driven engineering (MDE) (Chapter 5). MDE relies on generating code from a reusable domain-specific model of a system. However, the code generators used in MDE often generate relatively inefficient code.

Approach	Description
Application frameworks	Collections of abstract and concrete classes are adapted and extended to create application systems.
Application system integration	Two or more application systems are integrated to provide extended functionality.
Architectural patterns	Standard software architectures that support common types of application system are used as the basis of applications. Described in Chapters 6, 11, and 17.
Aspect-oriented software development	Shared components are woven into an application at different places when the program is compiled. Described in web Chapter 31.
Component-based software engineering	Systems are developed by integrating components (collections of objects) that conform to component-model standards. Described in Chapter 16.
Configurable application systems	Domain-specific systems are designed so that they can be configured to the needs of specific system customers.
Design patterns	Generic abstractions that occur across applications are represented as design patterns showing abstract and concrete objects and interactions. Described in Chapter 7.
ERP systems	Large-scale systems that encapsulate generic business functionality and rules are configured for an organization.
Legacy system wrapping	Legacy systems (Chapter 9) are "wrapped" by defining a set of interfaces and providing access to these legacy systems through these interfaces.
Model-driven engineering	Software is represented as domain models and implementation independent models, and code is generated from these models. Described in Chapter 5.
Program generators	A generator system embeds knowledge of a type of application and is used to generate systems in that domain from a user-supplied system model.
Program libraries	Class and function libraries that implement commonly used abstractions are available for reuse.
Service-oriented systems	Systems are developed by linking shared services, which may be externally provided. Described in Chapter 18.
Software product lines	An application type is generalized around a common architecture so that it can be adapted for different customers.
Systems of systems	Two or more distributed systems are integrated to create a new system. Described in Chapter 20.

Figure 15.4 Approaches that support software reuse

5. *The application domain* In many application domains, such as manufacturing and medical information systems, there are generic products that may be reused by configuring them to a local situation. This is one of the most effective approaches to reuse, and it is almost always cheaper to buy rather than build a new system.

Generator-based reuse

Generator-based reuse involves incorporating reusable concepts and knowledge into automated tools and providing an easy way for tool users to integrate specific code with this generic knowledge. This approach is usually most effective in domain-specific applications. Known solutions to problems in that domain are embedded in the generator system and selected by the user to create a new system.

http://software-engineering-book.com/web/generator-reuse/

6. *The platform on which the system will run* Some components models, such as .NET, are specific to Microsoft platforms. Similarly, generic application systems may be platform-specific, and you may only be able to reuse these if your system is designed for the same platform.

The range of available reuse techniques is such that, in most situations, there is the possibility of some software reuse. Whether or not reuse is achieved is often a managerial rather than a technical issue. Managers may be unwilling to compromise their requirements to allow reusable components to be used. They may not understand the risks associated with reuse as well as they understand the risks of original development. Although the risks of new software development may be higher, some managers may prefer known risks of development to unknown risks of reuse. To promote company-wide reuse, it may be necessary to introduce a reuse program that focuses on the creation of reusable assets and processes to facilitate reuse (Jacobsen, Griss, and Jonsson 1997).

15.2 Application frameworks

Early enthusiasts for object-oriented development suggested that one of the key benefits of using an object-oriented approach was that objects could be reused in different systems. However, experience has shown that objects are often too fine-grained and are often specialized for a particular application. It often takes longer to understand and adapt the object than to reimplement it. It has now become clear that object-oriented reuse is best supported in an object-oriented development process through larger-grain abstractions called frameworks.

As the name suggests, a framework is a generic structure that is extended to create a more specific subsystem or application. Schmidt et al. (Schmidt et al. 2004) define a framework to be

> *an integrated set of software artifacts (such as classes, objects and components) that collaborate to provide a reusable architecture for a family of related applications.*[†]

Frameworks provide support for generic features that are likely to be used in all applications of a similar type. For example, a user interface framework will provide support

[†]Schmidt, D. C., A. Gokhale, and B. Natarajan. 2004. "Leveraging Application Frameworks." ACM Queue 2 (5 (July/August)): 66–75. doi:10.1145/1016998.1017005.

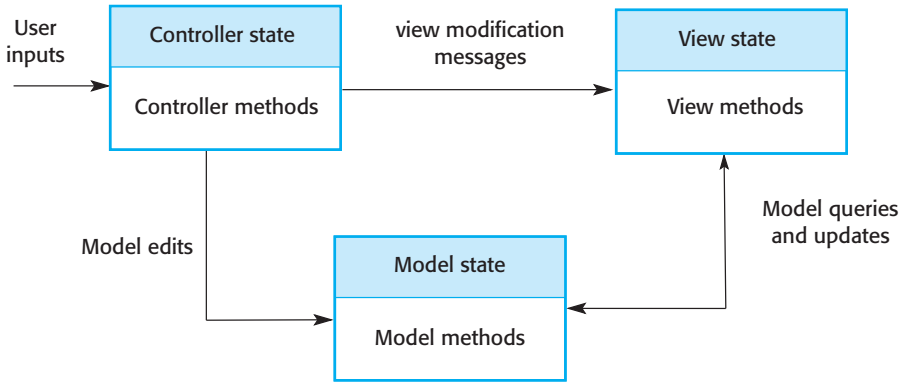


Figure 15.5 The Model-View-Controller pattern

for interface event handling and will include a set of widgets that can be used to construct displays. It is then left to the developer to specialize these by adding specific functionality for a particular application. For example, in a user interface framework, the developer defines display layouts that are appropriate to the application being implemented.

Frameworks support design reuse in that they provide a skeleton architecture for the application as well as the reuse of specific classes in the system. The architecture is implemented by the object classes and their interactions. Classes are reused directly and may be extended using features such as inheritance and polymorphism.

Frameworks are implemented as a collection of concrete and abstract object classes in an object-oriented programming language. Therefore, frameworks are language-specific. Frameworks are available in commonly used object-oriented programming languages such as Java, C#, and C++, as well as in dynamic languages such as Ruby and Python. In fact, a framework can incorporate other frameworks, where each framework is designed to support the development of part of the application. You can use a framework to create a complete application or to implement part of an application, such as the graphical user interface.

The most widely used application frameworks are web application frameworks (WAFs), which support the construction of dynamic websites. The architecture of a WAF is usually based on the Model-View-Controller (MVC) Composite pattern shown in Figure 15.5. The MVC pattern was originally proposed in the 1980s as an approach to GUI design that allowed for multiple presentations of an object and separate styles of interaction with each of these presentations. In essence, it separates the state from its presentation so that the state may be updated from each presentation.

An MVC framework supports the presentation of data in different ways and allows interaction with each of these presentations. When the data is modified through one of the presentations, the system model is changed and the controllers associated with each view update their presentation.

Frameworks are often implementations of design patterns, as discussed in Chapter 7. For example, an MVC framework includes the Observer pattern, the Strategy pattern, the Composite pattern, and a number of others that are discussed by Gamma et al. (Gamma et al. 1995). The general nature of patterns and their use of abstract and concrete classes allow for extensibility. Without patterns, frameworks would almost certainly be impractical.

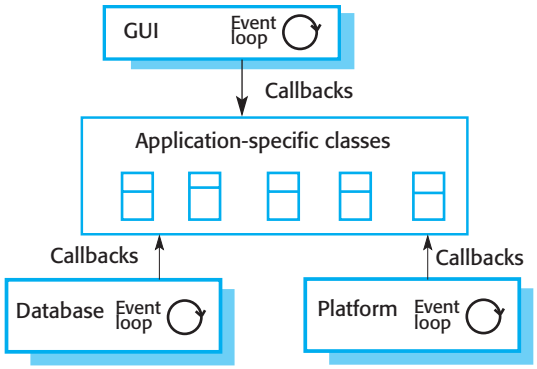


Figure 15.6 Inversion of control in frameworks

While each framework includes slightly different functionality, web application frameworks usually provide components and classes that support:

1. *Security* WAFs may include classes to help implement user authentication (login) and access control to ensure that users can only access permitted functionality in the system.
2. *Dynamic web pages* Classes are provided to help you define web page templates and to populate these dynamically with specific data from the system database.
3. *Database integration* Frameworks don't usually include a database but assume that a separate database, such as MySQL, will be used. The framework may include classes that provide an abstract interface to different databases.
4. *Session management* Classes to create and manage sessions (a number of interactions with the system by a user) are usually part of a WAF.
5. *User interaction* Web frameworks provide AJAX (Holdener 2008) and/or HTML5 support (Sarris 2013), which allows interactive web pages to be created. They may include classes that allow device-independent interfaces to be created, which adapt automatically to mobile phones and tablets.

To implement a system using a framework, you add concrete classes that inherit operations from abstract classes in the framework. In addition, you define "callbacks"—methods that are called in response to events recognized by the framework. The framework objects, rather than the application-specific objects, are responsible for control in the system. Schmidt et al. (Schmidt, Gokhale, and Natarajan 2004) call this "inversion of control."

In response to events from the user interface and database framework objects invoke "hook methods" that are then linked to user-provided functionality. The user-provided functionality defines how the application should respond to the event (Figure 15.6). For example, a framework will have a method that handles a mouse click from the environment. This method is called the hook method, which you must configure to call the appropriate application methods to handle the mouse click.

Fayad and Schmidt (Fayad and Schmidt 1997) discuss three other classes of framework:

1. *System infrastructure frameworks* support the development of system infrastructures such as communications, user interfaces, and compilers.
2. *Middleware integration frameworks* consist of a set of standards and associated object classes that support component communication and information exchange. Examples of this type of framework include Microsoft's .NET and Enterprise Java Beans (EJB). These frameworks provide support for standardized component models, as discussed in Chapter 16.
3. *Enterprise application frameworks* are concerned with specific application domains such as telecommunications or financial systems (Baumer et al. 1997). These embed application domain knowledge and support the development of end-user applications. These are not now widely used and have been largely superseded by software product lines.[†]

Applications that are constructed using frameworks can be the basis for further reuse through the concept of software product lines or application families. Because these applications are constructed using a framework, modifying family members to create instances of the system is often a straightforward process. It involves rewriting concrete classes and methods that you have added to the framework.

Frameworks are a very effective approach to reuse. However, they are expensive to introduce into software development processes as they are inherently complex and it can take several months to learn to use them. It can be difficult and expensive to evaluate available frameworks to choose the most appropriate one. Debugging framework-based applications is more difficult than debugging original code because you may not understand how the framework methods interact. Debugging tools may provide information about the reused framework components, which the developer does not understand.

15.3 Software product lines

When a company has to support a number of similar but not identical systems, one of the most effective approaches to reuse is to create a software product line. Hardware control systems are often developed using this approach to reuse as are domain-specific applications in areas such as logistics or medical systems. For example, a printer manufacturer has to develop printer control software, where there is a specific version of the product for each type of printer. These software versions have much in common, so it makes sense to create a core product (the product line) and adapt this for each printer type.

A software product line is a set of applications with a common architecture and shared components, with each application specialized to reflect specific customer requirements. The core system is designed so that it can be configured and adapted to

[†]Fayad, M. E., and D. C. Schmidt. 1997. "Object-Oriented Application Frameworks." Comm. ACM 40 (10): 32–38. doi:10.1145/262793.262798.

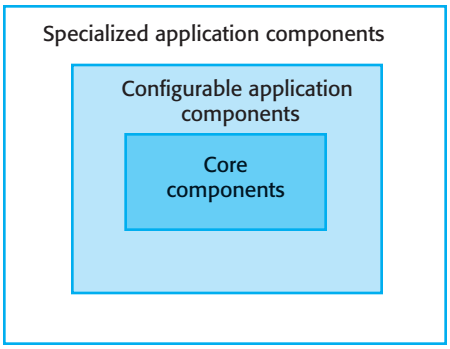


Figure 15.7 The organization of a base system for a product line

suit the needs of different customers or equipment. This may involve the configuration of some components, implementing additional components, and modifying some of the components to reflect new requirements.

Developing applications by adapting a generic version of the application means that a high proportion of the application code is reused in each system. Testing is simplified because tests for large parts of the application may also be reused, thus reducing the overall application development time. Engineers learn about the application domain through the software product line and so become specialists who can work quickly to develop new applications.

Software product lines usually emerge from existing applications. That is, an organization develops an application and then, when a similar system is required, informally reuses code from this in the new application. The same process is used as other similar applications are developed. However, change tends to corrupt application structure so, as more new instances are developed, it becomes increasingly difficult to create a new version. Consequently, a decision to design a generic product line may then be made. This involves identifying common functionality in product instances and developing a base application, which is then used for future development.

This base application (Figure 15.7) is designed to simplify reuse and reconfiguration. Generally, a base application includes:

1. Core components that provide infrastructure support. These are not usually modified when developing a new instance of the product line.
2. Configurable components that may be modified and configured to specialize them to a new application. Sometimes it is possible to reconfigure these components without changing their code by using a built-in component configuration language.
3. Specialized, domain-specific components some or all of which may be replaced when a new instance of a product line is created.

Application frameworks and software product lines have much in common. They both support a common architecture and components, and require new development to create a specific version of a system. The main differences between these approaches are as follows:

1. Application frameworks rely on object-oriented features such as inheritance and polymorphism to implement extensions to the framework. Generally, the framework

code is not modified, and the possible modifications are limited to whatever is supported by the framework. Software product lines are not necessarily created using an object-oriented approach. Application components are changed, deleted, or rewritten. There are no limits, in principle at least, to the changes that can be made.

2. Most application frameworks provide general support rather than domain-specific support. For example, there are application frameworks to create web-based applications. A software product line usually embeds detailed domain and platform information. For example, there could be a software product line concerned with web-based applications for health record management.

3. Software product lines are often control applications for equipment. For example, there may be a software product line for a family of printers. This means that the product line has to provide support for hardware interfacing. Application frameworks are usually software-oriented, and they do not usually include hardware interaction components.

4. Software product lines are made up of a family of related applications, owned by the same organization. When you create a new application, your starting point is often the closest member of the application family, not the generic core application.

If you are developing a software product line using an object-oriented programming language, then you may use an application framework as a basis for the system. You create the core of the product line by extending the framework with domain-specific components using its built-in mechanisms. There is then a second phase of development where versions of the system for different customers are created. For example, you can use a web-based framework to build the core of a software product line that supports web-based help desks. This "help desk product line" may then be further specialized to provide particular types of help desk support.

The architecture of a software product line often reflects a general, application-specific architectural style or pattern. For example, consider a product-line system that is designed to handle vehicle dispatching for emergency services. Operators of this system take calls about incidents, find the appropriate vehicle to respond to the incident, and dispatch the vehicle to the incident site. The developers of such a system may market versions of it for police, fire, and ambulance services.

This vehicle dispatching system is an example of a generic resource allocation and management architecture (Figure 15.8). Resource management systems use a database of available resources and include components to implement the resource allocation policy that has been decided by the company using the system. Users interact with a resource management system to request and release resources and to ask questions about resources and their availability.

You can see how this four-layer structure may be instantiated in Figure 15.9, which shows the modules that might be included in a vehicle dispatching system product line. The components at each level in the product-line system are as follows:

1. At the interaction level, components provide an operator display interface and an interface with the communications systems used.

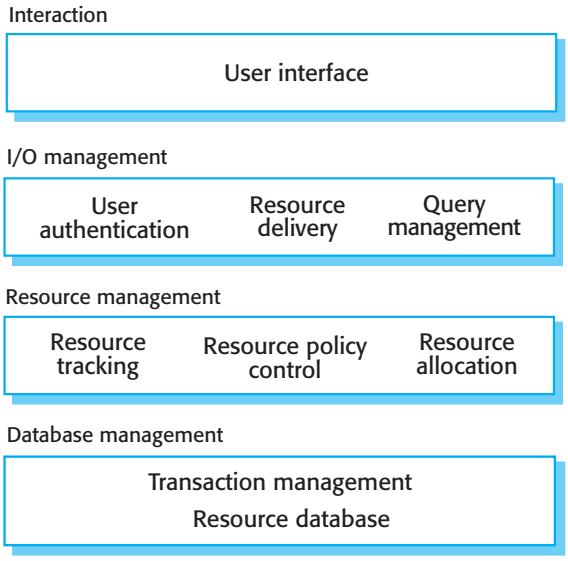


Figure 15.8 The architecture of a resource management system

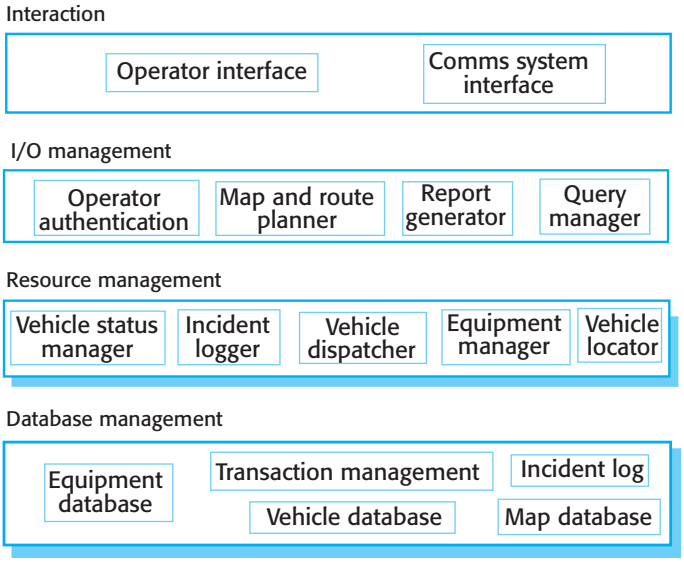


Figure 15.9 A product-line architecture of a vehicle dispatcher system

2. At the I/O management level (level 2), components handle operator authentication, generate reports of incidents and vehicles dispatched, support map output and route planning, and provide a mechanism for operators to query the system databases.
3. At the resource management level (level 3), components allow vehicles to be located and dispatched, update the status of vehicles and equipment, and log details of incidents.
4. At the database level, as well as the usual transaction management support, there are separate databases of vehicles, equipment, and maps.

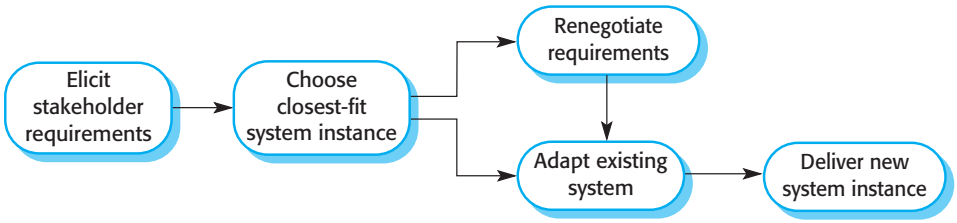


Figure 15.10 Product instance development

To create a new instance of this system, you may have to modify individual components. For example, the police have a large number of vehicles but a relatively small number of vehicle types. By contrast, the fire service has many types of specialized vehicles but relatively few vehicles. Therefore, when you are implementing a system for these different services, you may have to define a different vehicle database structure.

Various types of specialization of a software product line may be developed:

1. *Platform specialization* Versions of the application may be developed for different platforms. For example, versions of the application may exist for Windows, Mac OS, and Linux platforms. In this case, the functionality of the application is normally unchanged; only those components that interface with the hardware and operating system are modified.
2. *Environment specialization* Versions of the application may be created to handle different operating environments and peripheral devices. For example, a system for the emergency services may exist in different versions, depending on the communications hardware used by each service. For example, police radios may have built-in encryption that has to be used. The product-line components are changed to reflect the functionality and characteristics of the equipment used.
3. *Functional specialization* Versions of the application may be created for specific customers who have different requirements. For example, a library automation system may be modified depending on whether it is used in a public library, a reference library, or a university library. In this case, components that implement functionality may be modified and new components added to the system.
4. *Process specialization* The system may be adapted to cope with specific business processes. For example, an ordering system may be adapted to cope with a centralized ordering process in one company and with a distributed process in another.

Figure 15.10 shows the process for extending a software product line to create a new application. The activities in this process are:

1. *Elicit stakeholder requirements* You may start with a normal requirements engineering process. However, because a system already exists, you can demonstrate the system and have stakeholders experiment with it, expressing their requirements as modifications to the functions provided.

2. *Select the existing system that is the closest fit to the requirements* When creating a new member of a product line, you may start with the nearest product instance. The requirements are analyzed, and the family member that is the closest fit is chosen for modification.
3. *Renegotiate requirements* As more details of required changes emerge and the project is planned, some requirements may be renegotiated with the customer to minimize the changes that will have to be made to the base application.
4. *Adapt existing system* New modules are developed for the existing system, and existing system modules are adapted to meet the new requirements.
5. *Deliver new product family member* The new instance of the product line is delivered to the customer. Some deployment-time configuration may be required to reflect the particular environments where the system will be used. At this stage, you should document its key features so that it may be used as a basis for other system developments in the future.

When you create a new member of a product line, you may have to find a compromise between reusing as much of the generic application as possible and satisfying detailed stakeholder requirements. The more detailed the system requirements, the less likely it is that the existing components will meet these requirements. However, if stakeholders are willing to be flexible and to limit the system modifications that are required, you can usually deliver the system more quickly and at a lower cost.

Software product lines are designed to be reconfigurable. This reconfiguration may involve adding or removing components from the system, defining parameters and constraints for system components, and including knowledge of business processes. This configuration may occur at different stages in the development process:

1. *Design-time configuration* The organization that is developing the software modifies a common product-line core by developing, selecting, or adapting components to create a new system for a customer.
2. *Deployment-time configuration* A generic system is designed for configuration by a customer or consultants working with the customer. Knowledge of the customer's specific requirements and the system's operating environment is embedded in the configuration data used by the generic system.

When a system is configured at design time, the supplier starts with either a generic system or an existing product instance. By modifying and extending modules in this system, the supplier creates a specific system that delivers the required customer functionality. This usually involves changing and extending the source code of the system so that greater flexibility is possible than with deployment-time configuration.

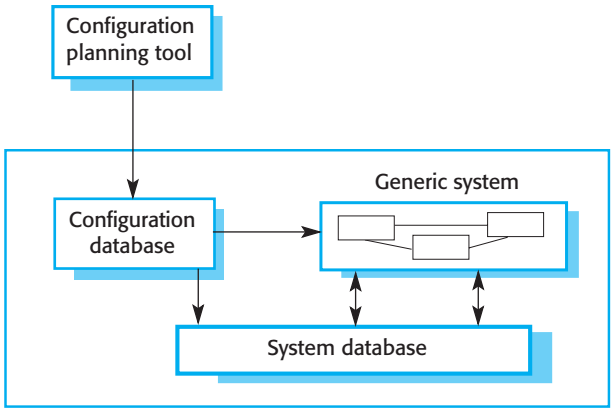


Figure 15.11 Deployment-time configuration

Design-time configuration is used when it is impossible to use the existing deployment-time configuration facilities in a system to develop a new system version. However, over time, when you have created several family members with comparable functionality, you may decide to refactor the core product line to include functionality that has been implemented in several application family members. You then make that new functionality configurable when the system is deployed.

Deployment-time configuration involves using a configuration tool to create a specific system configuration that is recorded in a configuration database or as a set of configuration files (Figure 15.11). The executing system, which may either run on a server or as a stand-alone system on a PC, consults this database when executing so that its functionality may be specialized to its execution context.

Several levels of deployment-time configuration may be provided in a system:

1. *Component selection*, where you select the modules in a system that provide the required functionality. For example, in a patient information system, you may select an image management component that allows you to link medical images (X-rays, CT scans, etc.) to the patient's medical record.
2. *Workflow and rule definition*, where you define workflows (how information is processed, stage by stage), and validation rules that should apply to information entered by users or generated by the system.
3. *Parameter definition*, where you specify the values of specific system parameters that reflect the instance of the application that you are creating. For example, you may specify the maximum length of fields for data input by a user or the characteristics of hardware attached to the system.

Deployment-time configuration can be very complex, and for large systems, it may take several months to configure and test a system for a customer. Large configurable systems may support the configuration process by providing software tools, such as planning tools, to support the configuration process. I discuss deployment-time configuration further in Section 15.4.1. This discussion covers the reuse of application systems that have to be configured to work in different operational environments.

15.4 Application system reuse

An application system product is a software system that can be adapted to the needs of different customers without changing the source code of the system. Application systems are developed by a system vendor for a general market; they are not specially developed for an individual customer. These system products are sometimes known as COTS (Commercial Off-the Shelf System) products. However, the term "COTS" is mostly used in military systems, and I prefer to call these system products *application systems*.

Virtually all desktop software for business and many server-based systems are application systems. This software is designed for general use, so it includes many features and functions. It therefore has the potential to be reused in different environments and as part of different applications. Torchiano and Morisio (Torchiano and Morisio 2004) also discovered that open-source products were often used without change and without looking at the source code.

Application system products are adapted by using built-in configuration mechanisms that allow the functionality of the system to be tailored to specific customer needs. For example, in a hospital patient record system, separate input forms and output reports might be defined for different types of patients. Other configuration features may allow the system to accept plug-ins that extend functionality or check user inputs to ensure that they are valid.

This approach to software reuse has been very widely adopted by large companies since the late 1990s, as it offers significant benefits over customized software development:

1. As with other types of reuse, more rapid deployment of a reliable system may be possible.
2. It is possible to see what functionality is provided by the applications, and so it is easier to judge whether or not they are likely to be suitable. Other companies may already use the applications, so experience of the systems is available.
3. Some development risks are avoided by using existing software. However, this approach has its own risks, as I discuss below.
4. Businesses can focus on their core activity without having to devote a lot of resources to IT systems development.
5. As operating platforms evolve, technology updates may be simplified as these are the responsibility of the application system vendor rather than the customer.

Of course, this approach to software engineering has its own problems:

1. Requirements usually have to be adapted to reflect the functionality and mode of operation of the off-the-shelf application system. This can lead to disruptive changes to existing business processes.

Configurable application systems	Application system integration
Single product that provides the functionality required by a customer	Several different application systems are integrated to provide customized functionality
Based on a generic solution and standardized processes	Flexible solutions may be developed for customer processes
Development focus is on system configuration	Development focus is on system integration
System vendor is responsible for maintenance	System owner is responsible for maintenance
System vendor provides the platform for the system	System owner provides the platform for the system

Figure 15.12 Individual and integrated application systems

2. The application system may be based on assumptions that are practically impossible to change. The customer must therefore adapt its business to reflect these assumptions.
3. Choosing the right application system for an enterprise can be a difficult process, especially as many of these systems are not well documented. Making the wrong choice means that it may be impossible to make the new system work as required.
4. There may be a lack of local expertise to support systems development. Consequently, the customer has to rely on the vendor and external consultants for development advice. This advice may be geared to selling products and services, with insufficient time taken to understand the real needs of the customer.
5. The system vendor controls system support and evolution. It may go out of business, be taken over, or make changes that cause difficulties for customers.

Application systems may be used as individual systems or in combination, where two or more systems are integrated. Individual systems consist of a generic application from a single vendor that is configured to customer requirements. Integrated systems involve integrating the functionality of individual systems, often from different vendors, to create a new application system. Figure 15.12 summarizes the differences between these different approaches. I discuss application system integration in Section 15.4.2.

15.4.1 Configurable application systems

Configurable application systems are generic application systems that may be designed to support a particular business type, business activity, or, sometimes, a complete business enterprise. For example, a system produced for dentists may handle appointments, reminders, dental records, patient recall, and billing. At a larger scale, an Enterprise Resource Planning (ERP) system may support the manufacturing, ordering, and customer relationship management processes in a large company.

Domain-specific application systems, such as systems to support a business function (e.g., document management), provide functionality that is likely to be required by a range of potential users. However, they also incorporate built-in assumptions about how

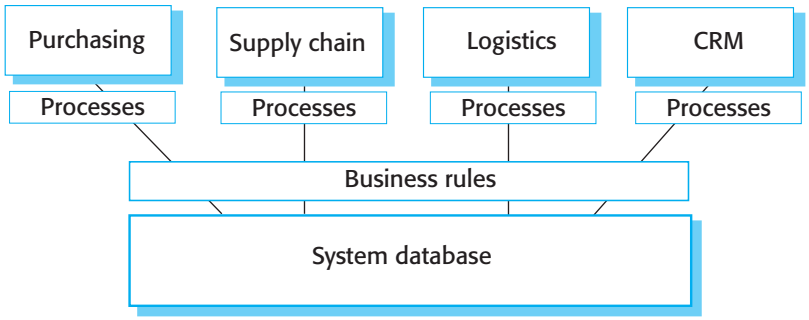


Figure 15.13 The architecture of an ERP system

users work, and these assumptions may cause problems in specific situations. For example, a system to support student registration in a university may assume that students will be registered for one degree at one university. However, if universities collaborate to offer joint degrees, then it may be practically impossible to represent this detail in the system.

Enterprise Resource Planning (ERP) systems, such as those produced by SAP and Oracle, are large-scale, integrated systems designed to support business practices such as ordering and invoicing, inventory management, and manufacturing scheduling (Monk and Wagner 2013). The configuration process for these systems involves gathering detailed information about the customer's business and business processes, and embedding this information in a configuration database. This often requires detailed knowledge of configuration notations and tools and is usually carried out by consultants working alongside system customers.

A generic ERP system includes a number of modules that may be composed in different ways to create a system for a customer. The configuration process involves choosing which modules are to be included, configuring these individual modules, defining business processes and business rules, and defining the structure and organization of the system database. A model of the overall architecture of an ERP system that supports a range of business functions is shown in Figure 15.13.

The key features of this architecture are as follows:

1. A number of modules to support different business functions. These are large grain modules that may support entire departments or divisions of the business. In the example shown in Figure 15.13, the modules that have been selected for inclusion in the system are a module to support purchasing; a module to support supply chain management; a logistics module to support the delivery of goods; and a customer relationship management (CRM) module to maintain customer information.
2. A defined set of business process models, associated with each module, which relate to activities in that module. For example, the ordering process model may define how orders are created and approved. This will specify the roles and activities involved in placing an order.
3. A common database that maintains information about all related business functions. Thus, it should not be necessary to replicate information, such as customer details, in different parts of the business.

4. A set of business rules that apply to all data in the database. Therefore, when data is input from one function, these rules should ensure that it is consistent with the data required by other functions. For example, a business rule may require that all expense claims have to be approved by someone more senior than the person making the claim.

ERP systems are used in almost all large companies to support some or all of their functions. They are, therefore, a very widely used form of software reuse. The obvious limitation of this approach to reuse is that the functionality of the customer's application is restricted to the functionality of the ERP system's built-in modules. If a company needs additional functionality, it may have to develop a separate add-on system to provide this functionality.

Furthermore, the buyer company's processes and operations have to be defined in the ERP system's configuration language. This language embeds the understanding of business processes as seen by the system vendor, and there may be a mismatch between these assumptions and the concepts and processes used in the customer's business. A serious mismatch between the customer's business model and the system model used by the ERP system makes it highly probable that the ERP system will not meet the customer's real needs (Scott 1999).

For example, in an ERP system that was sold to a university, a fundamental system concept was the notion of a customer. In this system, a customer was an external agent that bought goods and services from a supplier. This concept caused great difficulties when configuring the system. Universities do not really have customers. Rather, they have customer-type relationships with a range of people and organizations such as students, research funding agencies, and educational charities. None of these relationships is compatible with a customer relationship where a person or business buys products or services from another. In this particular case, it took several months to resolve this mismatch, and the final solution only partially met the university's requirements.

ERP systems usually require extensive configuration to adapt them to the requirements of each organization where they are installed. This configuration may involve:

1. Selecting the required functionality from the system, for example, by deciding what modules should be included.
2. Establishing a data model that defines how the organization's data will be structured in the system database.
3. Defining business rules that apply to that data.
4. Defining the expected interactions with external systems.
5. Designing the input forms and the output reports generated by the system.
6. Designing new business processes that conform to the underlying process model supported by the system.
7. Setting parameters that define how the system is deployed on its underlying platform.

Once the configuration settings are completed, the new system is then ready for testing. Testing is a major problem when systems are configured rather than programmed using a conventional language. There are two reasons for this:

1. Test automation may be difficult or impossible. There may be no easy access to an API that can be used by testing frameworks such as JUnit, so the system has to be tested manually by testers inputting test data to the system. Furthermore, systems are often specified informally, so defining test cases may be difficult without a lot of help from end-users.
2. Systems errors are often subtle and specific to business processes. The application system or ERP system is a reliable platform, so technical system failures are rare. The problems that occur are often due to misunderstandings between those configuring the system and user stakeholders. System testers without detailed knowledge of the end-user processes cannot detect these errors.

15.4.2 Integrated application systems

Integrated application systems include two or more application systems or, sometimes, legacy systems. You may use this approach when no single application system meets all of your needs or when you wish to integrate a new application system with systems that you are already using. The component systems may interact through their APIs or service interfaces if these are defined. Alternatively, they may be composed by connecting the output of one system to the input of another or by updating the databases used by the applications.

To develop integrated application systems, you have to make a number of design choices:

1. *Which individual application systems offer the most appropriate functionality?* Typically, several system products will be available, which can be combined in different ways. If you don't already have experience with a particular application system, it can be difficult to decide which product is the most suitable.
2. *How will data be exchanged?* Different systems normally use unique data structures and formats. You have to write adaptors that convert from one representation to another. These adaptors are runtime systems that operate alongside the constituent application systems.
3. *What features of a product will actually be used?* Individual application systems may include more functionality than you need, and functionality may be duplicated across different products. You have to decide which features in what product are most appropriate for your requirements. If possible, you should also deny access to unused functionality because this can interfere with normal system operation.

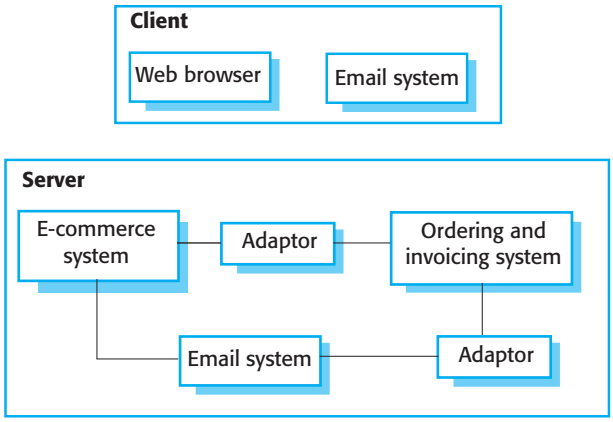


Figure 15.14 An integrated procurement system

Consider the following scenario as an illustration of application system integration. A large organization intends to develop a procurement system that allows staff to place orders from their desk. By introducing this system across the organization, the company estimates that it can save $5 million per year. By centralizing buying, the new procurement system can ensure that orders are always made from suppliers who offer the best prices and should reduce the administration associated with orders. As with manual systems, the system involves choosing the goods available from a supplier, creating an order, having the order approved, sending the order to a supplier, receiving the goods, and confirming that payment should be made.

The company has a legacy ordering system that is used by a central procurement office. This order processing software is integrated with an existing invoicing and delivery system. To create the new ordering system, the legacy system is integrated with a web-based e-commerce platform and an email system that handles communications with users. The structure of the final procurement system is shown in Figure 15.14.

This procurement system should be a client–server system with standard web browsing and email systems used on the client. On the server, the e-commerce platform has to integrate with the existing ordering system through an adaptor. The e-commerce system has its own format for orders, confirmations of delivery, and so forth, and these have to be converted into the format used by the ordering system. The e-commerce system uses the email system to send notifications to users, but the ordering system was never designed for this purpose. Therefore, another adaptor has to be written to convert the notifications from the ordering system into email messages.

Months, sometimes years, of implementation effort can be saved, and the time to develop and deploy a system can be drastically reduced by integrating existing application systems. The procurement system described above was implemented and deployed in a very large company in nine months. It had originally been estimated that it would take three years to develop a procurement system in Java that could be integrated with the legacy ordering system.

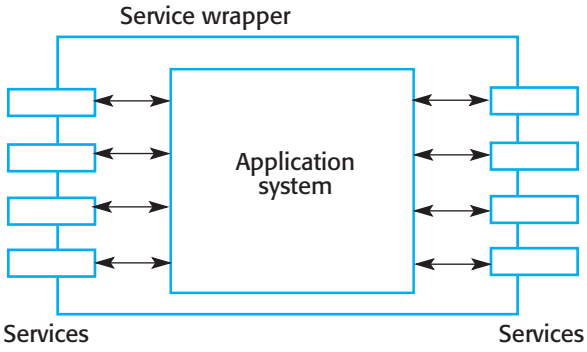


Figure 15.15
Application wrapping

Application system integration can be simplified if a service-oriented approach is used. Essentially, a service-oriented approach means allowing access to the application system's functionality through a standard service interface, with a service for each discrete unit of functionality. Some applications may offer a service interface, but sometimes this service interface has to be implemented by the system integrator. Essentially, you have to program a wrapper that hides the application and provides externally visible services (Figure 15.15). This approach is particularly valuable for legacy systems that have to be integrated with newer application systems.

In principle, integrating application systems is the same as integrating any other component. You have to understand the system interfaces and use them exclusively to communicate with the software; you have to trade off specific requirements against rapid development and reuse; and you have to design a system architecture that allows the application systems to operate together.

However, the fact that these products are usually large systems in their own right, and are often sold as separate standalone systems, introduces additional problems. Boehm and Abts (Boehm and Abts 1999) highlight four important system integration problems:

1. *Lack of control over functionality and performance* Although the published interface of a product may appear to offer the required facilities, the system may not be properly implemented or may perform poorly. The product may have hidden operations that interfere with its use in a specific situation. Fixing these problems may be a priority for the system integrator but may not be of real concern for the product vendor. Users may simply have to find workarounds to problems if they wish to reuse the application system.

2. *Problems with system interoperability* It is sometimes difficult to get individual application systems to work together because each system embeds its own assumptions about how it will be used. Garlan et al. (Garlan, Allen, and Ockerbloom 1995), reporting on their experience integrating four application systems, found that three of these products were event-based but that each used a different model of events. Each system assumed that it had exclusive access to the event queue. As a consequence, integration was very difficult. The project

required five times as much effort as originally predicted. The schedule was extended to two years rather than the predicted six months.

In a retrospective analysis of their work 10 years later, Garlan et al. (Garlan, Allen, and Ockerbloom 2009) concluded that the integration problems that they discovered had not been solved. Torchiano and Morisio (Torchiano and Morisio 2004) found that lack of compliance with standards in many application systems meant that integration was more difficult than anticipated.

3. *No control over system evolution* Vendors of application systems make their own decisions on system changes, in response to market pressures. For PC products in particular, new versions are often produced frequently and may not be compatible with all previous versions. New versions may have additional unwanted functionality, and previous versions may become unavailable and unsupported.

4. *Support from system vendors* The level of support available from system vendors varies widely. Vendor support is particularly important when problems arise as developers do not have access to the source code and detailed documentation of the system. While vendors may commit to providing support, changing market and economic circumstances may make it difficult for them to deliver this commitment. For example, a system vendor may decide to discontinue a product because of limited demand, or they may be taken over by another company that does not wish to support the products that have been acquired.

Boehm and Abts reckon that, in many cases, the cost of system maintenance and evolution may be greater for integrated application systems. The above difficulties are life-cycle problems; they don't just affect the initial development of the system. The further removed the people involved in the system maintenance become from the original system developers, the more likely it is that difficulties will arise with the integrated system.

KEY POINTS

- There are many different ways to reuse software. These range from the reuse of classes and methods in libraries to the reuse of complete application systems.
- The advantages of software reuse are lower costs, faster software development, and lower risks. System dependability is increased. Specialists can be used more effectively by concentrating their expertise on the design of reusable components.
- Application frameworks are collections of concrete and abstract objects that are designed for reuse through specialization and the addition of new objects. They usually incorporate good design practice through design patterns.

- Software product lines are related applications that are developed from one or more base applications. A generic system is adapted and specialized to meet specific requirements for functionality, target platform, or operational configuration.
- Application system reuse is concerned with the reuse of large-scale, off-the-shelf systems. These provide a lot of functionality, and their reuse can radically reduce costs and development time. Systems may be developed by configuring a single, generic application system or by integrating two or more application systems.
- Potential problems with application system reuse include lack of control over functionality, performance, and system evolution; the need for support from external vendors; and difficulties in ensuring that systems can interoperate.

FURTHER READING

"Overlooked Aspects of COTS-Based Development." An interesting article that discusses a survey of developers using a COTS-based approach, and the problems that they encountered. (M. Torchiano and M. Morisio, *IEEE Software*, 21 (2), March–April 2004) http://dx.doi.org/10.1109/MS.2004.1270770

CRUISE—Component Reuse in Software Engineering. This e-book covers a wide range of reuse topics, including case studies, component-based reuse, and reuse processes. However, its coverage of application system reuse is limited. (L. Nascimento et al., 2007) http://www.academia.edu/179616/C.R.U.I.S.E_-_Component_Reuse_in_Software_Engineering

"Construction by Configuration: A New Challenge for Software Engineering." In this invited paper, I discuss the problems and difficulties of constructing a new application by configuring existing systems. (I. Sommerville, *Proc. 19th Australian Software Engineering Conference*, 2008) http://dx.doi.org/10.1109/ASWEC.2008.75

"Architectural Mismatch: Why Reuse Is Still So Hard." This article looks back on an earlier paper that discussed the problems of reusing and integrating a number of application systems. The authors concluded that, although some progress has been made, there were still problems in conflicting assumptions made by the designers of the individual systems. (D. Garlan et al., *IEEE Software*, 26 (4), July–August 2009) http://dx.doi.org//10.1109/MS.2009.86

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-reuse/

EXERCISES

15.1. What major technical and nontechnical factors hinder software reuse? Do you personally reuse much software and, if not, why not?

15.2. List the benefits of software reuse and explain why the expected lifetime of the software should be considered when planning reuse.

15.3. How does the base application's design in the product line simplify reuse and reconfiguration?

15.4. Explain what is meant by "inversion of control" in application frameworks. Explain why this approach could cause problems if you integrated two separate systems that were originally created using the same application framework.

15.5. Using the example of the weather station system described in Chapters 1 and 7, suggest a product-line architecture for a family of applications that are concerned with remote monitoring and data collection. You should present your architecture as a layered model, showing the components that might be included at each level.

15.6. Most desktop software, such as word processing software, can be configured in a number of different ways. Examine software that you regularly use and list the configuration options for that software. Suggest difficulties that users might have in configuring the software. Microsoft Office (or one of its open-source alternatives) is a good example to use for this exercise.

15.7. Why have many large companies chosen ERP systems as the basis for their organizational information system? What problems may arise when deploying a large-scale ERP system in an organization?

15.8. What are the significant benefits offered by the application system reuse approach when compared with the custom software development approach?

15.9. Explain why adaptors are usually needed when systems are constructed by integrating application systems. Suggest three practical problems that might arise in writing adaptor software to link two application systems.

15.10. The reuse of software raises a number of copyright and intellectual property issues. If a customer pays a software contractor to develop a system, who has the right to reuse the developed code? Does the software contractor have the right to use that code as a basis for a generic component? What payment mechanisms might be used to reimburse providers of reusable components? Discuss these issues and other ethical issues associated with the reuse of software.

REFERENCES

Baumer, D., G. Gryczan, R. Knoll, C. Lilienthal, D. Riehle, and H. Zullighoven. 1997. "Framework Development for Large Systems." *Comm. ACM* 40 (10): 52–59. doi:10.1145/262793.262804.

Boehm, B., and C. Abts. 1999. "COTS Integration: Plug and Pray?" *Computer* 32 (1): 135–138. doi:10.1109/2.738311.

Fayad, M.E., and D.C. Schmidt. 1997. "Object-Oriented Application Frameworks." *Comm. ACM* 40 (10): 32–38. doi:10.1145/262793.262798.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Garlan, D., R. Allen, and J. Ockerbloom. 1995. "Architectural Mismatch: Why Reuse Is So Hard." *IEEE Software* 12 (6): 17–26. doi:10.1109/52.469757.

———. 2009. "Architectural Mismatch: Why Reuse Is Still so Hard." *IEEE Software* 26 (4): 66–69. doi:10.1109/MS.2009.86.

Holdener, A.T. 2008. *Ajax: The Definitive Guide*. Sebastopol, CA: O'Reilly and Associates.

Jacobsen, I., M. Griss, and P. Jonsson. 1997. *Software Reuse*. Reading, MA: Addison-Wesley.

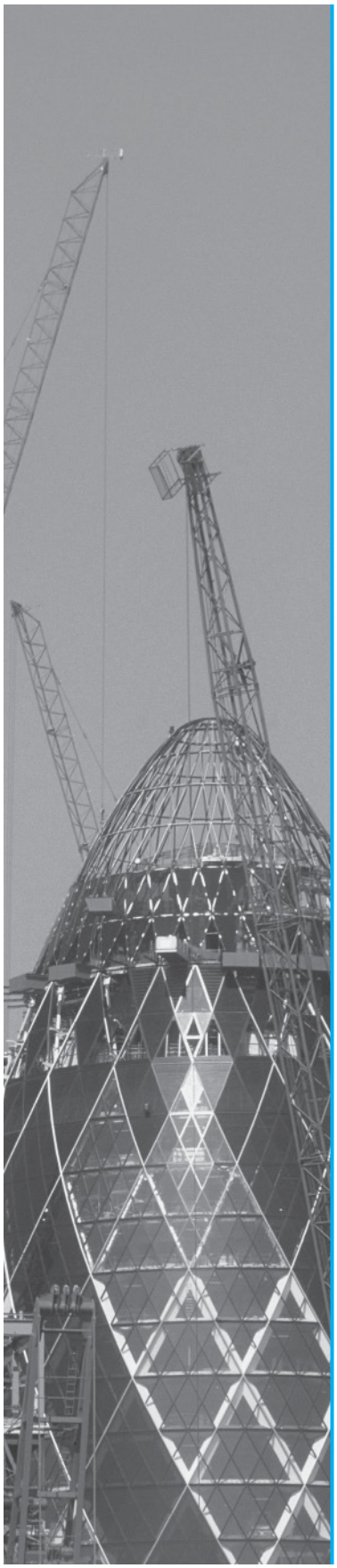
Monk, E., and B. Wagner. 2013. *Concepts in Enterprise Resource Planning, 4th ed.* Independence, KY: CENGAGE Learning.

Sarris, S. 2013. *HTML5 Unleashed*. Indianapolis, IN: Sams Publishing.

Schmidt, D. C., A. Gokhale, and B. Natarajan. 2004. "Leveraging Application Frameworks." *ACM Queue* 2 (5 (July/August)): 66–75. doi:10.1145/1016998.1017005.

Scott, J. E. 1999. "The FoxMeyer Drug's Bankruptcy: Was It a Failure of ERP." In *Proc. Association for Information Systems 5th Americas Conf. on Information Systems*. Milwaukee, WI. http://www.uta.edu/faculty/weltman/OPMA5364TW/FoxMeyer.pdf

Torchiano, M., and M. Morisio. 2004. "Overlooked Aspects of COTS-Based Development." *IEEE Software* 21 (2): 88–93. doi:10.1109/MS.2004.1270770.



16

Component-based software engineering

Objectives

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

- understand what is meant by a software component that may be included in a program as an executable element;
- understand the key elements of software component models and the support provided by middleware for these models;
- be aware of the key activities in the component-based software engineering (CBSE) process for reuse and the CBSE process with reuse;
- understand three different types of component composition and some of the problems that have to be resolved when components are composed to create new components or systems.

Contents

16.1 Components and component models
16.2 CBSE processes
16.3 Component composition

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 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, which usually meant that you had to have the component source code. Selling or distributing objects as individual reusable components was therefore 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. Component-based software engineering is the process of defining, implementing, and integrating or composing these loosely coupled, independent components into systems.

CBSE has become as an important software development approach for large-scale enterprise systems, with demanding performance and security requirements. Customers are demanding secure and dependable software that is delivered and deployed more quickly. The only way that these demands can be met is to build software by reusing existing 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 the need to change other parts of the system.
2. Component standards that define interfaces and so 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 on the functionality of available components.

Component-based development embodies good software engineering practice. It often makes sense to design a system using components, even if you have to develop

Problems with CBSE

CBSE is now a mainstream approach to software engineering and is widely used when creating new systems. However, when used as an approach to reuse, problems include component trustworthiness, component certification, requirements compromises, and prediction of the properties of components, especially when they are integrated with other components.

http://software-engineering-book.com/web/cbse-problems/

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 component providing 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 technology-specific "standards" were introduced to support this view of a component, including Sun's Enterprise Java Beans (EJB), Microsoft's COM and .NET, and CORBA's CCM (Lau and Wang 2007).

Unfortunately, the companies involved in proposing standards could not agree on a single standard for components, thereby limiting the impact of this approach to software reuse. It is 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 is a type of component-based software engineering. It uses a simpler notion of a component than that originally proposed in CBSE,

where components were executable routines that were included in larger systems. Each system that used a component embedded its own version of that component. Service-oriented approaches are gradually replacing CBSE with embedded components as an approach to systems development. In this chapter, I discuss the use of CBSE with embedded components; service-oriented software engineering is covered in Chapter 18.

16.1 Components and component models

The software reuse community generally agrees 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 (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 standards-based so that a software unit that conforms to these standards is a component. Szyperski (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 were developed around the idea of a component as an element that is embedded in a system, rather than a service that is referenced by the system. However, they are equally applicable to service components.

Szyperski also states that a component has no externally observable state; that is, 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. While stateless components are certainly simpler to use, in some systems 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. I think that, if we combine these proposals, we get a more rounded description of a reusable component. Figure 16.1 shows what I consider to be the essential characteristics of a component as used in CBSE.

[†]Councill, W. T., and G. T. Heineman. 2001. "Definition of a Software Component and Its Elements." In Component-Based Software Engineering, edited by G T Heineman and W T Councill, 5–20. Boston: Addison Wesley.

[‡]Szyperski, C. 2002. Component Software: Beyond Object-Oriented Programming, 2nd Ed. Harlow, UK: Addison Wesley.

Component characteristic	Description
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 that 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.
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.
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.

Figure 16.1 Component characteristics

A useful way of thinking about a component is as a provider of one or more services, even if the component is embedded rather than implemented as a service. When a system needs something to be done, 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 system used in a public library might provide a search service that allows users to search the library catalog. A component that converts from one graphical format to another (e.g., TIFF to JPEG) provides a data conversion service and so on.

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.

In principle, all components have two related interfaces, as shown in Figure 16.2. These interfaces reflect the services that the component provides and the services that the component requires to operate correctly:

1. The "provides" interface defines the services provided by the component. This interface is the component API. It defines the methods that can be called by a user

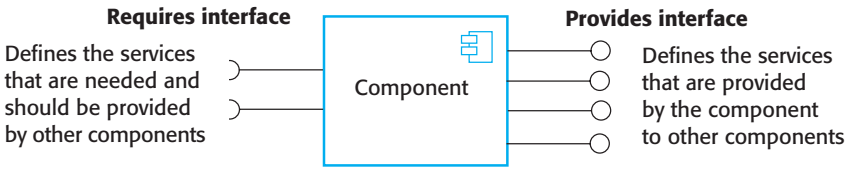


Figure 16.2 Component interfaces

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.

2. The "requires" interface specifies the services that other components in the system must provide if a component is to operate correctly. If these services 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 16.3 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 them 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 interface, accessed through sensorManagement. This interface has been designed to connect to different types of sensors so that it does not include specific sensor operations such as Test and provideReading. 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 parameter 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 may be connected to a sensor (Figure 16.12).

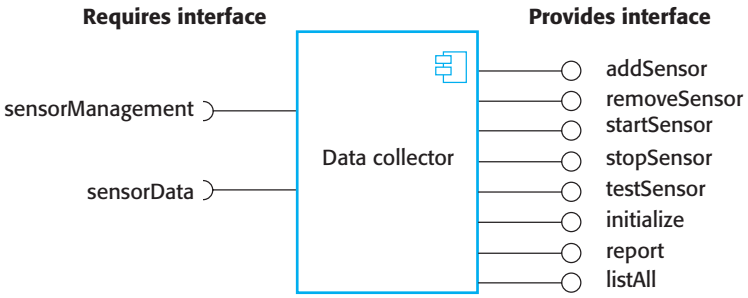


Figure 16.3 A model of a data collector component

Components 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://software-engineering-book.com/web/components-and-objects/

Components are accessed using remote procedure calls (RPCs). Each component has a unique identifier and, using this name, may be called from another computer. The called component uses the same mechanism to access the "required" components that are defined in its interface.

An important difference between a component as an external service and a component as a program element accessed using a remote procedure call is that services are completely independent entities. They do not have an explicit "requires" interface. Of course, they do require other components to support their operation, but these are provided internally. Other programs can use services without the need to implement any additional support required by the service.

16.1.1 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. For service components, the most important component model is the Web Service models, and for embedded components, widely used models include the 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 (Weinreich and Sametinger 2001). I summarize these model elements in Figure 16.4. 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, interface specification uses XML-based languages as discussed in Chapter 18; EJB is Java-specific, so Java is used as the interface definition language; in .NET, interfaces are defined using Microsoft's Common

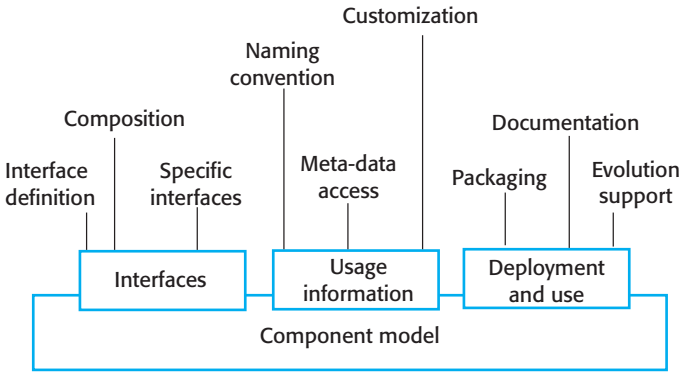


Figure 16.4 Basic elements of a component model

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 via RPCs, 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 16.3) 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 routines. 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 define the component documentation that should be produced. This is used to find the component and to decide whether it is appropriate.

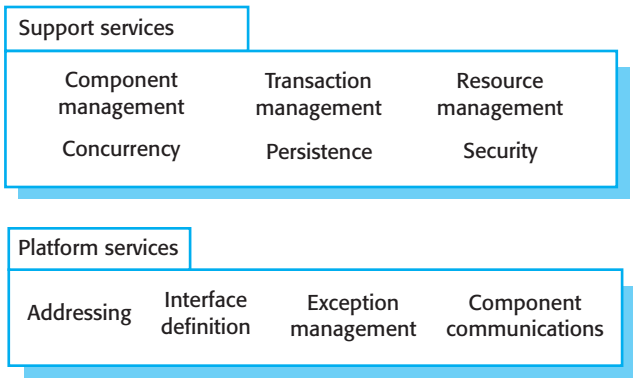


Figure 16.5 Middleware services defined in a component model

For components that are executable routines rather than external services, the component model defines the services to be provided by the middleware that supports the executing components. Weinreich and Sametinger 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 16.5 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 many different components are likely to require. 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.

Middleware implements the common component services and provides interfaces to them. 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. Conceptually, when you add a component to the container, 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. 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.

The services associated with a component model have much in common with the facilities provided by object-oriented frameworks, which I discussed in Chapter 15. Although the services provided may not be as comprehensive, framework services are often more efficient than container-based services. As a consequence, some people think that it is best to use frameworks such as SPRING (Wheeler and White 2013) for Java development rather than the fully-featured component model in EJB.

16.2 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 16.6 (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 process 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 16.6 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.

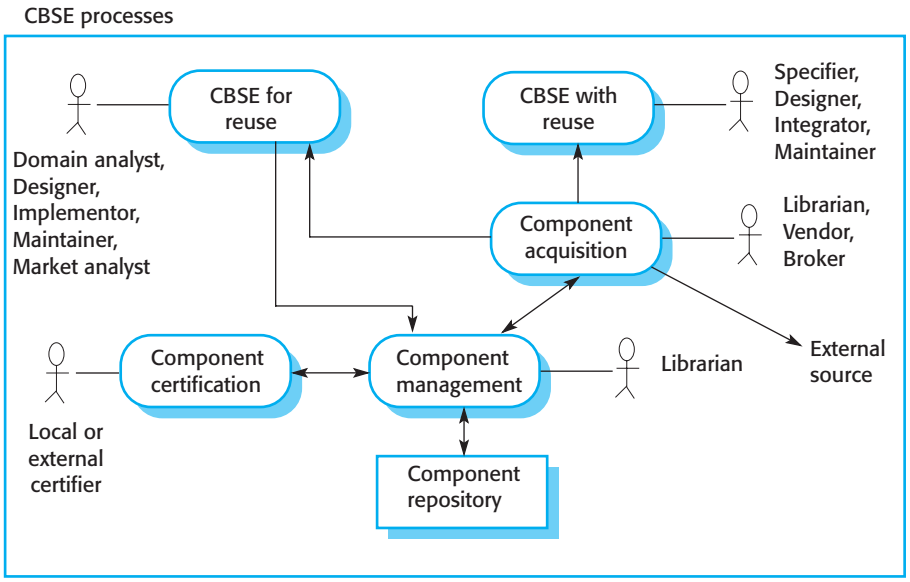


Figure 16.6 CBSE processes

2. *Component management* is concerned with managing a company's reusable components, ensuring that they are properly catalogued, stored, and made available for reuse.
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.

16.2.1 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 off-the-shelf components is uncommon.

Consequently, CBSE for reuse is mostly used within organizations that have made a commitment to reuse-driven software engineering. These companies have a base of internally developed components that can be reused. However, these internally developed components may not be 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. 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 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 and component validation, and of 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 difficult one. In principle, components should not handle exceptions themselves because each application will have its own requirements for exception management. Rather, the component should define what exceptions can arise and should publish these exceptions 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 process:

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.

You therefore have to take a pragmatic approach to component exception handling. Common technical exceptions, where recovery is important for the functioning of the component, should be handled locally. These exceptions and how they are handled

should be documented with the component. Other exceptions that are related to the business function of the component should be passed to the calling component for handling.

Mili et al. (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. 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.

Whether or not a component is reusable depends on its application domain, functionality, and generality. If the domain is a general one and the component implements standard functionality in that domain, then it is more likely to be reusable. As you add generality to a component, you increase its reusability because it can be applied in a wider range of environments. Unfortunately, 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, a trade-off between the reusability and understandability 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. Reusability adds complexity and hence reduces component understandability. This makes it more difficult and time consuming to decide whether a component is suitable for reuse. Because of the time involved in understanding a reusable component, it is sometimes more cost-effective to reimplement a simpler component with the specific functionality that is required.

A potential source of components is legacy systems. As I discussed in Chapter 9, legacy systems are systems that fulfill an important business function but are written using obsolete software technologies. As a result, 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 may be significantly less than the cost of reimplementing the legacy system.

Once you have developed and tested a reusable component or service, it 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 GitHub or Sourceforge. If it is intended for use in a company, then you may use an internal repository system.

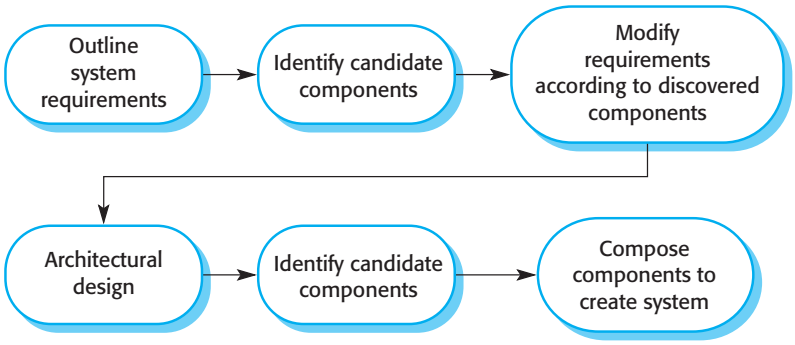


Figure 16.7 CBSE with reuse

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 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 process can be expensive, and so many companies simply leave testing and quality checking to the component developers.

16.2.2 CBSE with reuse

The successful reuse of components requires a development process tailored to including reusable components in the software being developed. 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 16.7 shows the principal activities within that process. Some of these activities, 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 as follows:

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 description of the 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 by the reusable components. 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. Apparently, usable components may turn out

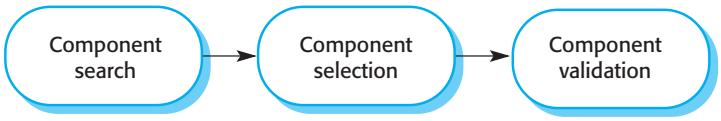


Figure 16.8 The component identification process

to be unsuitable or may not work properly with other chosen components. You may have to find alternatives to these components. Further requirements changes may therefore be necessary, depending on the functionality of these components.

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. Jacobsen et al. (Jacobsen, Griss, and Jonsson 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 predetermined. As I discussed in Chapter 6, you also establish the high-level architecture 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 16.8. Initially, your focus should be on search and selection. You need to convince yourself that components are 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 within your company or from trusted suppliers. There are few component vendors, so you are most likely to be looking for components that have been developed in your own organization or in the repositories of open-source software that are available. 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, GitHub, or Google Code, to see if source code for the component that you need is available.

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 more complex. There will not be a clear mapping of requirements onto components. You may find that several components have to be integrated to meet a

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. The designers 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. 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 runtime 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.

This illustrates an important problem with software reuse. Software may be based on assumptions about the context where the system will be used, and these assumptions may not be valid in a different situation.

More information about this failure is available at: http://software-engineering-book.com/case-studies/ariane5/

Figure 16.9 An example of validation failure with reused software

specific requirement or group of requirements. You therefore have to decide which of these 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 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, reused 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. While this may have no effect on the application reusing the component, it can slow down the component, cause it to produce surprising results or, in exceptional cases, cause serious system failures. Figure 16.9 summarizes a situation where the failure of a reused software system, which had unnecessary functionality, led to catastrophic system 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 a specific application environment and, naturally, embed assumptions about that environment. These assumptions are rarely documented, so when the component is reused, it is impossible to develop tests to check if the assumptions are still valid. If you are reusing a component in a new environment, you may not discover the embedded environmental assumptions until you use the component in an operational system.

16.3 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. You can compose components in several different ways, as shown in Figure 16.10. 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* In a sequential composition, you create a new component from two 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 but are called by the external application. This type of composition may be used with embedded or service components.

   Some extra glue code may be 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 "glue code" transforms these outputs to be of the form expected by component B.

2. *Hierarchical composition* This type of composition occurs when one component calls directly on the services provided by another component. That is, component A calls component B. 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 services accessed over the web.

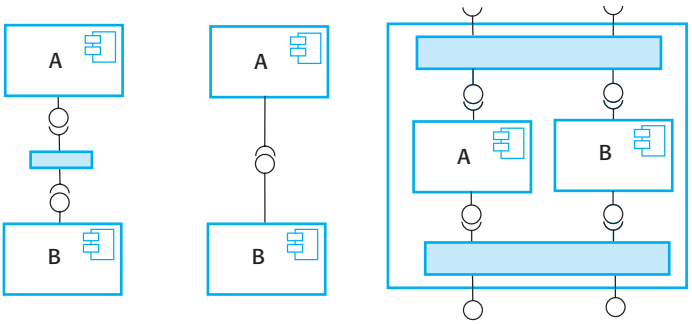


Figure 16.10 Types of component composition

3. *Additive composition* This occurs when two or more components are put together (added) to create a new component, which combines their functionality. The "provides" interface and "requires" interface of the new component are a combination of the corresponding interfaces in components A and B. The components are called separately through the external interface of the composed component and may be called in any order. A and B are not dependent and do not call each other. This type of composition may be used with embedded or service components.

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. In Figure 16.11, the location parameter returned by addressFinder is incompatible with the parameters required by the displayMap and printMap methods in mapDB.
2. *Operation incompatibility* The names of the operations in the provides and "requires" interfaces are different. This is a further incompatibility between the components shown in Figure 16.11.
3. *Operation incompleteness* The "provides" interface of a component is a subset of the "requires" interface of another component, or vice versa.

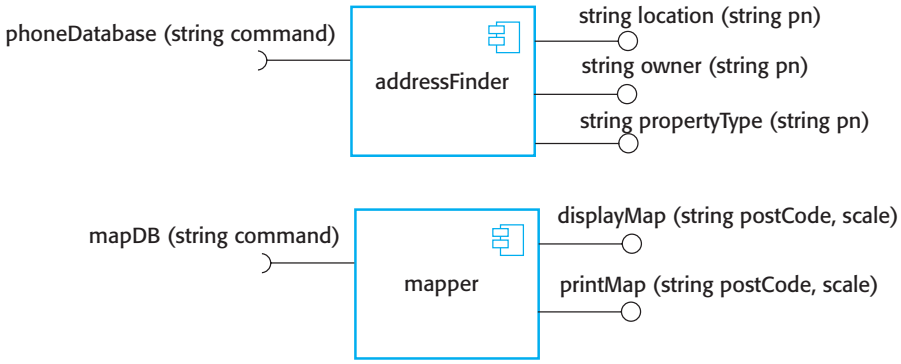


Figure 16.11 Components with incompatible interfaces

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 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 simple components shown in Figure 16.11, 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.

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 process:

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

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

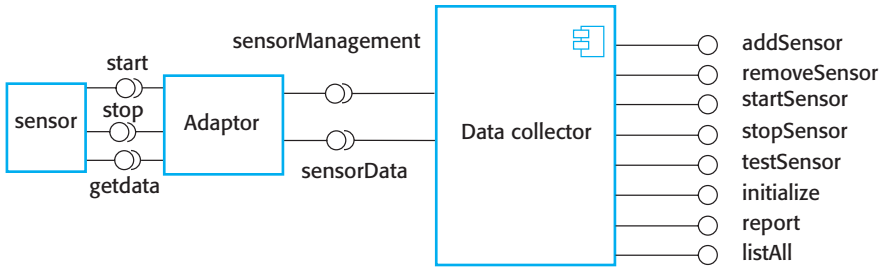


Figure 16.12 An adaptor linking a data collector and a sensor

between the "provides" interface and "requires" interface of the components in the composition. I have illustrated the use of an adaptor in Figure 16.12 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 adaptor 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 16.12, 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 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 16.13. These components are used to implement a system that downloads images from a camera and stores them in a photograph library. The system user can provide additional information to describe and catalog the photograph. To avoid clutter, I have not shown all interface

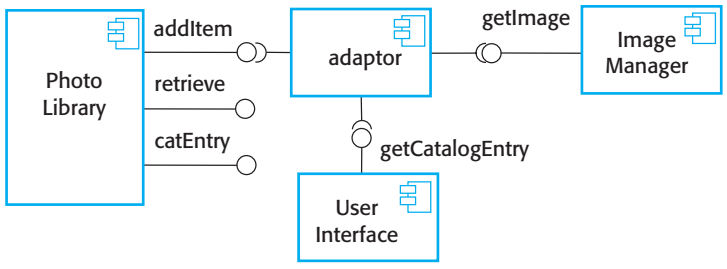


Figure 16.13 Photo library composition

```
— 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 16.14 The OCL description of the Photo Library 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 catalog 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 16.14 is part of the description of the interface of **Photo Library** that adds information to the informal description.

Figure 16.14 shows pre- and postconditions 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 true before a method has executed; and that must be true after a method has executed. These are invariants, preconditions, and postconditions. 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 primarily used in model-based software engineering to add semantic information to UML models. The 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 runtime system. Meyer suggests that using Design by Contract is essential if we are to develop trusted components (Meyer 2003).

Figure 16.14 shows the specification for the addItem and delete methods in Photo Library. The method being specified is indicated by the keyword context and the pre- and postconditions by the keywords pre and post. The preconditions 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 postconditions 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 catalog using that identifier, you get back the catalog entry that you made.

The specification of delete provides further information. The precondition 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 catalog 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

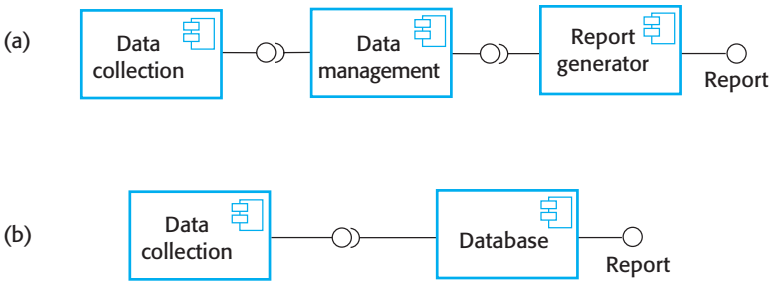


Figure 16.15 Data collection and report generation components

can evolve as requirements change. You may have to take trade-offs into account for component decisions:

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 properties include performance and dependability. You can only assess these properties once the complete system is implemented.

Unfortunately, in many situations the solutions to the composition problems may conflict. For example, consider a situation such as that illustrated in Figure 16.15, 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 likely to be 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 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 so that each component has a clearly defined role. Ideally, component roles should not overlap. However, it may be cheaper to buy one multifunctional component rather than two or three separate components. Furthermore, dependability or performance penalties may be incurred 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 executable routines 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 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 comprehensive discussion of commercial and research component models 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

"Software Components Beyond Programming: From Routines to Services." This is the opening article in a special issue of the magazine that includes several articles on software components. This article discusses the evolution of components and how service-oriented components are replacing executable program routines. (I. Crnkovic, J. Stafford, and C. Szyperski, *IEEE Software*, 28 (3), May/June 2011) http://dx.doi.org/10.1109/MS.2011.62

Object Constraint Language (OCL) Tutorial. A good introduction to the use of the object-constraint language. (J. Cabot, 2012) http://modeling-languages.com/ocl-tutorial/

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-reuse/

A more detailed discussion of the Ariane5 accident:

http://software-engineering-book.com/case-studies/ariane5/

EXERCISES

16.1. What are the design principles underlying the CBSE that support the construction of understandable and maintainable software?

16.2. 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.

16.3. In a reusable component, what are the critical characteristics that are emphasized when the component is viewed as a service?

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

16.5. 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.

16.6. What are the essential differences between CBSE with reuse and software processes for original software development?

16.7. Design the "provides" interface and the "requires" interface of a reusable component that may be used to represent a patient in the Mentcare system that I introduced in Chapter 1.

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

16.9. 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 dispatched to the incident.

16.10. 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

Councill, W. T., and G. T. Heineman. 2001. "Definition of a Software Component and Its Elements." In *Component-Based Software Engineering*, edited by G. T. Heineman and W. T. Councill, 5–20. Boston: Addison-Wesley.

Jacobsen, I., M. Griss, and P. Jonsson. 1997. *Software Reuse*. Reading, MA: Addison-Wesley.

Kotonya, G. 2003. "The CBSE Process: Issues and Future Visions." In *2nd CBSEnet Workshop*. Budapest, Hungary. http://miro.sztaki.hu/projects/cbsenet/budapest/presentations/Gerald-CBSEProcess.ppt

Lau, K-K., and Z. Wang. 2007. "Software Component Models." *IEEE Trans. on Software Eng.* 33 (10): 709–724. doi:10.1109/TSE.2007.70726.

Meyer, B. 1992. "Applying Design by Contract." *IEEE Computer* 25 (10): 40–51. doi:10.1109/2.161279.

———. 2003. "The Grand Challenge of Trusted Components." In *Proc. 25th Int. Conf. on Software Engineering*. Portland, OR: IEEE Press. doi:10.1109/ICSE.2003.1201252.

Mili, H., A. Mili, S. Yacoub, and E. Addy. 2002. *Reuse-Based Software Engineering*. New York: John Wiley & Sons.

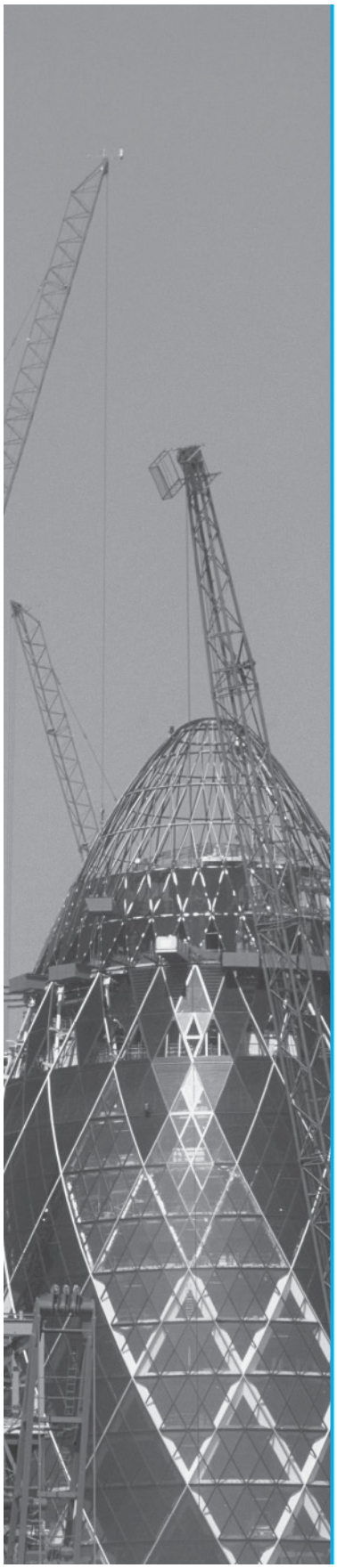
Pope, A. 1997. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Harlow, UK: Addison-Wesley.

Szyperski, C. 2002. *Component Software: Beyond Object-Oriented Programming, 2nd ed.* Harlow, UK: Addison-Wesley.

Warmer, J., and A. Kleppe. 2003. *The Object Constraint Language: Getting Your Models Ready for MDA*. Boston: Addison-Wesley.

Weinreich, R., and J. Sametinger. 2001. "Component Models and Component Services: Concepts and Principles." In *Component-Based Software Engineering*, edited by G. T. Heineman and W. T. Councill, 33–48. Boston: Addison-Wesley.

Wheeler, W., and J. White. 2013. *Spring in Practice*. Greenwich, CT: Manning Publications.



17 Distributed software engineering

Objectives

The objective of this chapter is to introduce distributed systems engineering and distributed systems architectures. When you have read this chapter, you will:

- know the key issues that have to be considered when designing and implementing distributed software systems;
- understand the client–server computing model and the layered architecture of client–server systems;
- have been introduced to commonly used patterns for distributed systems architectures and know the types of system for which each architectural pattern is applicable;
- understand the notion of software as a service, providing web-based access to remotely deployed application systems.

Contents

17.1 Distributed systems
17.2 Client–server computing
17.3 Architectural patterns for distributed systems
17.4 Software as a service

Most computer-based systems are now distributed systems. A distributed system is one involving several computers rather than a single application running on a single machine. Even apparently self-contained applications on a PC or laptop, such as image editors, are distributed systems. They execute on a single computer system but often rely on remote cloud systems for update, storage, and other services. Tanenbaum and Van Steen (Tanenbaum and Van Steen 2007) define a distributed system to be "a collection of independent computers that appears to the user as a single coherent system."†

When you are designing a distributed system, there are specific issues that have to be taken into account simply because the system is distributed. These issues arise because different parts of the system are running on independently managed computers and because the characteristics of the network, such as latency and reliability, may have to be considered in your design.

Coulouris et al. (Coulouris et al. 2011) identify the five benefits of developing systems as distributed systems:

1. *Resource sharing* A distributed system allows the sharing of hardware and software resources—such as disks, printers, files, and compilers—that are associated with computers on a network.
2. *Openness* Distributed systems are normally open systems—systems designed around standard Internet protocols so that equipment and software from different vendors can be combined.
3. *Concurrency* In a distributed system, several processes may operate at the same time on separate computers on the network. These processes may (but need not) communicate with each other during their normal operation.
4. *Scalability* In principle at least, distributed systems are scalable in that the capabilities of the system can be increased by adding new resources to cope with new demands on the system. In practice, the network linking the individual computers in the system may limit the system scalability.
5. *Fault tolerance* The availability of several computers and the potential for replicating information means that distributed systems can be tolerant of some hardware and software failures (see Chapter 11). In most distributed systems, a degraded service can be provided when failures occur; complete loss of service only occurs when there is a network failure.‡

Distributed systems are inherently more complex than centralized systems. This makes them more difficult to design, implement, and test. It is harder to understand the emergent properties of distributed systems because of the complexity of the interactions between system components and system infrastructure. For example, rather than being dependent on the execution speed of one processor, system performance

†Tanenbaum, A. S., and M. Van Steen. 2007. Distributed Systems: Principles and Paradigms, 2nd Ed. Upper Saddle River, NJ: Prentice-Hall.

‡Coulouris, G., J. Dollimore, T. Kindberg, and G. Blair. 2011. Distributed Systems: Concepts and Design, 5th Edition. Harlow, UK.: Addison Wesley.

depends on network bandwidth, network load, and the speed of other computers that are part of the system. Moving resources from one part of the system to another can significantly affect the system's performance.

Furthermore, as all users of the WWW know, distributed systems are unpredictable in their response. Response time depends on the overall load on the system, its architecture, and the network load. As all of these factors may change over a short time, the time taken to respond to a user request may change significantly from one request to another.

The most important developments that have affected distributed software systems in the past few years are service-oriented systems and the advent of cloud computing, delivering infrastructure, platforms, and software as a service. In this chapter, I focus on general issues of distributed systems, and in Section 17.4 I cover the idea of software as a service. In Chapter 18, I discuss other aspects of service-oriented software engineering.

17.1 Distributed systems

As I discussed in the introduction to this chapter, distributed systems are more complex than systems that run on a single processor. This complexity arises because it is practically impossible to have a top-down model of control of these systems. The nodes in the system that deliver functionality are often independent systems that are managed and controlled by their owners. There is no single authority in charge of the entire distributed system. The network connecting these nodes is also a separately managed system. It is a complex system in its own right and cannot be controlled by the owners of systems using the network. There is, therefore, an inherent unpredictability in the operation of distributed systems that has to be taken into account when you are designing a system.

Some of the most important design issues that have to be considered in distributed systems engineering are:

1. *Transparency* To what extent should the distributed system appear to the user as a single system? When is it useful for users to understand that the system is distributed?
2. *Openness* Should a system be designed using standard protocols that support interoperability, or should more specialized protocols be used? Although standard network protocols are now universally used, this is not the case for higher levels of interaction, such as service communication.
3. *Scalability* How can the system be constructed so that it is scalable? That is, how can the overall system be designed so that its capacity can be increased in response to increasing demands made on the system?
4. *Security* How can usable security policies be defined and implemented that apply across a set of independently managed systems?
5. *Quality of service* How should the quality of service that is delivered to system users be specified, and how should the system be implemented to deliver an acceptable quality of service to all users.
6. *Failure management* How can system failures be detected, contained (so that they have minimal effects on other components in the system), and repaired?

CORBA—Common Object Request Broker Architecture

CORBA was proposed as a specification for a middleware system in the 1990s by the Object Management Group. It was intended as an open standard that would allow the development of middleware to support distributed component communications and execution, as well as provide a set of standard services that could be used by these components.

Several implementations of CORBA were produced, but the system was not widely adopted. Users preferred proprietary systems such as those from Microsoft or Oracle, or they moved to service-oriented architectures.

http://software-engineering-book.com/web/corba/

In an ideal world, the fact that a system is distributed would be transparent to users. Users would see the system as a single system whose behavior is not affected by the way that the system is distributed. In practice, this is impossible to achieve because there is no central control over the system as a whole. As a result, individual computers in a system may behave differently at different times. Furthermore, because it always takes a finite length of time for signals to travel across a network, network delays are unavoidable. The length of these delays depends on the location of resources in the system, the quality of the user's network connection, and the network load.

To make a distributed system transparent (i.e., conceal its distributed nature), you have to hide the underlying distribution. You create abstractions that hide the system resources so that the location and implementation of these resources can be changed without having to change the distributed application. Middleware (discussed in Section 17.1.2) is used to map the logical resources referenced by a program onto the actual physical resources and to manage resource interactions.

In practice, it is impossible to make a system completely transparent, and users, generally, are aware that they are dealing with a distributed system. You may therefore decide that it is best to expose the distribution to users. They can then be prepared for some of the consequences of distribution such as network delays and remote node failures.

Open distributed systems are built according to generally accepted standards. Components from any supplier can therefore be integrated into the system and can interoperate with the other system components. At the networking level, openness is now taken for granted, with systems conforming to Internet protocols, but at the component level, openness is still not universal. Openness implies that system components can be independently developed in any programming language and, if these conform to standards, they will work with other components.

The CORBA standard (Pope 1997), developed in the 1990s, was intended to be the universal standard for open distributed systems, However, the CORBA standard never achieved a critical mass of adopters. Rather, many companies preferred to develop systems using proprietary standards for components from companies such as Sun (now Oracle) and Microsoft. These provided better implementations and support software and better long-term support for industrial protocols.

Web service standards (discussed in Chapter 18) for service-oriented architectures were developed to be open standards. However, these standards have met with significant resistance because of their perceived inefficiency. Many developers of service-based systems have opted instead for so-called RESTful protocols because

these have an inherently lower overhead than web service protocols. The use of RESTful protocols is not standardized.

The scalability of a system reflects its ability to deliver high-quality service as demands on the system increase. The three dimensions of scalability are size, distribution, and manageability.

1. *Size* It should be possible to add more resources to a system to cope with increasing numbers of users. Ideally, then, as the number of users increases, the system should increase in size automatically to handle the increased number of users.
2. *Distribution* It should be possible to geographically disperse the components of a system without degrading its performance. As new components are added, it should not matter where these are located. Large companies can often make use of computing resources in their different facilities around the world.
3. *Manageability* It should be possible to manage a system as it increases in size, even if parts of the system are located in independent organizations. This is one of the most difficult challenges of scale as it involves managers communicating and agreeing on management policies. In practice, the manageability of a system is often the factor that limits the extent to which it can be scaled.

Changing the size of a system may involve either scaling up or scaling out. Scaling up means replacing resources in the system with more powerful resources. For example, you may increase the memory in a server from 16 Gb to 64 Gb. Scaling out means adding more resources to the system (e.g., an extra web server to work alongside an existing server). Scaling out is often more cost-effective than scaling up, especially now that cloud computing makes it easy to add or remove servers from a system. However, this only provides performance improvements when concurrent processing is possible.

I have discussed general security issues and issues of security engineering in Part 2 of this book. When a system is distributed, attackers may target any of the individual system components or the network itself. If a part of the system is successfully attacked, then the attacker may be able to use this as a "back door" into other parts of the system.

A distributed system must defend itself against the following types of attack:

1. *Interception*, where an attacker intercepts communications between parts of the system so that there is a loss of confidentiality.
2. *Interruption*, where system services are attacked and cannot be delivered as expected. Denial-of-service attacks involve bombarding a node with illegitimate service requests so that it cannot deal with valid requests.
3. *Modification*, where an attacker gains access to the system and changes data or system services.
4. *Fabrication*, where an attacker generates information that should not exist and then uses this information to gain some privileges. For example, an attacker may generate a false password entry and use this to gain access to a system.

The major difficulty in distributed systems is establishing a security policy that can be reliably applied to all of the components in a system. As I discussed in Chapter 13, a security policy sets out the level of security to be achieved by a system. Security mechanisms, such as encryption and authentication, are used to enforce the security policy. The difficulties in a distributed system arise because different organizations may own parts of the system. These organizations may have mutually incompatible security policies and security mechanisms. Security compromises may have to be made in order to allow the systems to work together.

The quality of service (QoS) offered by a distributed system reflects the system's ability to deliver its services dependably and with a response time and throughput that are acceptable to its users. Ideally, the QoS requirements should be specified in advance and the system designed and configured to deliver that QoS. Unfortunately, this is not always practicable for two reasons:

1. It may not be cost-effective to design and configure the system to deliver a high quality of service under peak load. The peak demands may mean that you need many extra servers than normal to ensure that response times are maintained. This problem has been lessened by the advent of cloud computing where cloud servers may be rented from a cloud provider for as long as they are required. As demand increases, extra servers can be automatically added.
2. The quality-of-service parameters may be mutually contradictory. For example, increased reliability may mean reduced throughput, as checking procedures are introduced to ensure that all system inputs are valid.

Quality of service is particularly important when the system is dealing with time-critical data such as sound or video streams. In these circumstances, if the quality of service falls below a threshold value then the sound or video may become so degraded that it is impossible to understand. Systems dealing with sound and video should include quality of service negotiation and management components. These should evaluate the QoS requirements against the available resources and, if these are insufficient, negotiate for more resources or for a reduced QoS target.

In a distributed system, it is inevitable that failures will occur, so the system has to be designed to be resilient to these failures. Failure is so ubiquitous that one flippant definition of a distributed system suggested by Leslie Lamport, a prominent distributed systems researcher, is:

> *You know that you have a distributed system when the crash of a system that you've never heard of stops you getting any work done.*[†]

This is even truer now that more and more systems are executing in the cloud. Failure management involves applying the fault-tolerance techniques discussed in Chapter 11. Distributed systems should therefore include mechanisms for discovering whether a component of the system has failed, should continue to deliver as many services as possible in spite of that failure, and, as far as possible, should automatically

[†]Leslie Lamport, in Ross J. Anderson, Security Engineering: A Guide to Building Dependable Distributed Systems (2nd ed.), Wiley (April 14, 2008).

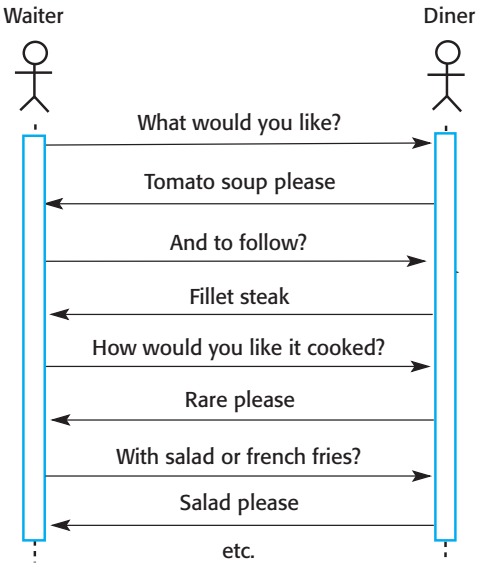


Figure 17.1 Procedural interaction between a diner and a waiter

recover from the failure. One important benefit of cloud computing is that it has dramatically reduced the cost of providing redundant system components.

17.1.1 Models of interaction

Two fundamental types of interaction may take place between the computers in a distributed computing system: procedural interaction and message-based interaction. Procedural interaction involves one computer calling on a known service offered by some other computer and waiting for that service to be delivered. Message-based interaction involves the "sending" computer defining information about what is required in a message, which is then sent to another computer. Messages usually transmit more information in a single interaction than a procedure call to another machine.

To illustrate the difference between procedural and message-based interaction, consider a situation where you are ordering a meal in a restaurant. When you have a conversation with the waiter, you are involved in a series of synchronous, procedural interactions that define your order. You make a request, the waiter acknowledges that request, you make another request, which is acknowledged, and so on. This is comparable to components interacting in a software system where one component calls methods from other components. The waiter writes down your order along with the order of other people with you. He or she then passes this order, which includes details of everything that has been ordered, to the kitchen to prepare the food. Essentially, the waiter is passing a message to the kitchen staff, defining the food to be prepared. This is message-based interaction.

I have illustrated this kind of interaction in Figure 17.1, which shows the synchronous ordering process as a series of calls, and in Figure 17.2, which shows a hypothetical XML message that defines an order made by the table of three people. The difference between these forms of information exchange is clear. The waiter takes the order as a series of

```
<starter>
   <dish name = “soup” type = “tomato” />
   <dish name = “soup” type = “fish” />
   <dish name = “pigeon salad” />
</starter>
<main course>
   <dish name = “steak” type = “sirloin” cooking = “medium” />
   <dish name = “steak” type = “fillet” cooking = “rare” />
   <dish name = “sea bass”>
</main>
<accompaniment>
   <dish name = “french fries” portions = “2” />
   <dish name = “salad” portions = “1” />
</accompaniment>
```

Figure 17.2
Message-based interaction between a waiter and the kitchen staff

interactions, with each interaction defining part of the order. However, the waiter has a single interaction with the kitchen where the message defines the complete order.

Procedural communication in a distributed system is usually implemented using remote procedure calls (RPCs). In an RPC, components have globally unique names (such as a URL). Using that name, a component can call on the services offered by another component as if it was a local procedure or method. System middleware intercepts this call and passes it on to a remote component. This carries out the required computation and, via the middleware, returns the result to the calling component. In Java, remote method invocations (RMIs) are remote procedure calls.

Remote procedure calls require a “stub” for the called procedure to be accessible on the computer that is initiating the call. This stub defines the interface of the remote procedure. The stub is called, and it translates the procedure parameters into a standard representation for transmission to the remote procedure. Through the middleware, it then sends the request for execution to the remote procedure. The remote procedure uses library functions to convert the parameters into the required format, carries out the computation, and then returns the results via the “stub” that is representing the caller.

Message-based interaction normally involves one component creating a message that details the services required from another component. This message is sent to the receiving component via the system middleware. The receiver parses the message, carries out the computations, and creates a message for the sending component with the required results. This is then passed to the middleware for transmission to the sending component.

A problem with the RPC approach to interaction is that both the caller and the callee need to be available at the time of the communication, and they must know how to refer to each other. In essence, an RPC has the same requirements as a local procedure or method call. By contrast, in a message-based approach, unavailability can be tolerated. If the system component that is processing the message is unavailable, the message simply stays in a queue until the receiver comes back online. Furthermore, it is not necessary for the sender to know the name of the message receiver and vice versa. They simply communicate with the middleware, which is responsible for ensuring that messages are passed to the appropriate system.

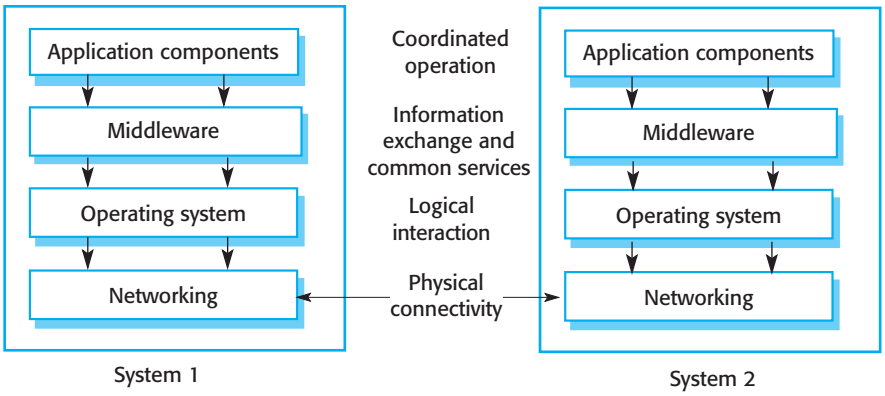


Figure 17.3 Middleware in a distributed system

17.1.2 Middleware

The components in a distributed system may be implemented in different programming languages and may execute on different types of processors. Models of data, information representation, and protocols for communication may all be different. A distributed system therefore requires software that can manage these diverse parts and ensure that they can communicate and exchange data.

The term *middleware* is used to refer to this software—it sits in the middle between the distributed components of the system. This concept is illustrated in Figure 17.3, which shows that middleware is a layer between the operating system and application programs. Middleware is normally implemented as a set of libraries, which are installed on each distributed computer, plus a runtime system to manage communications.

Bernstein (Bernstein 1996) describes types of middleware that are available to support distributed computing. Middleware is general-purpose software that is usually bought off-the-shelf rather than written specially by application developers. Examples of middleware include software for managing communications with databases, transaction managers, data converters, and communication controllers.

In a distributed system, middleware provides two distinct types of support:

1. *Interaction support*, where the middleware coordinates interactions between different components in the system. The middleware provides location transparency in that it isn't necessary for components to know the physical locations of other components. It may also support parameter conversion if different programming languages are used to implement components, event detection, communication, and so on.

2. *The provision of common services*, where the middleware provides reusable implementations of services that may be required by several components in the distributed system. By using these common services, components can easily interoperate and provide user services in a consistent way.

I have already given examples of the interaction support that middleware can provide in Section 17.1.1. You use middleware to support remote procedure and remote method calls, message exchange, and so forth.

Common services are those services that may be required by different components irrespective of the functionality of these components. As I discussed in Chapter 16, these may include security services (authentication and authorization), notification and naming services, and transaction management services. For distributed components, you can think of these common services as being provided by a middleware container; for services, they are provided through shared libraries. You then deploy your component, and it can access and use these common services.

17.2 Client–server computing

Distributed systems that are accessed over the Internet are organized as client–server systems. In a client–server system, the user interacts with a program running on their local computer, such as a web browser or app on a mobile device. This interacts with another program running on a remote computer, such as a web server. The remote computer provides services, such as access to web pages, which are available to external clients. This client–server model, as I discussed in Chapter 6, is a general architectural model of an application. It is not restricted to applications distributed across several machines. You can also use it as a logical interaction model where the client and the server run on the same computer.

In a client–server architecture, an application is modeled as a set of services that are provided by servers. Clients may access these services and present results to end-users. Clients need to be aware of the servers that are available but don't have to know anything about other clients. Clients and servers are separate processes, as shown in Figure 17.4. This figure illustrates a situation in which there are four servers (s1–s4) that deliver different services. Each service has a set of associated clients that access these services.

Figure 17.4 shows client and server processes rather than processors. It is normal for several client processes to run on a single processor. For example, on your PC, you may run a mail client that downloads mail from a remote mail server. You may also run a web browser that interacts with a remote web server and a print client that sends documents to a remote printer. Figure 17.5 shows a possible arrangement where the 12 logical clients shown in Figure 17.4 are running on six computers. The four server processes are mapped onto two physical server computers.

Several different server processes may run on the same processor, but, often, servers are implemented as multiprocessor systems in which a separate instance of the server process runs on each machine. Load-balancing software distributes requests for service from clients to different servers so that each server does the same amount of work. This allows a higher volume of transactions with clients to be handled, without degrading the response to individual clients.

Client–server systems depend on there being a clear separation between the presentation of information and the computations that create and process that information. Consequently, you should design the architecture of distributed client–server systems so that they are structured into several logical layers, with clear interfaces

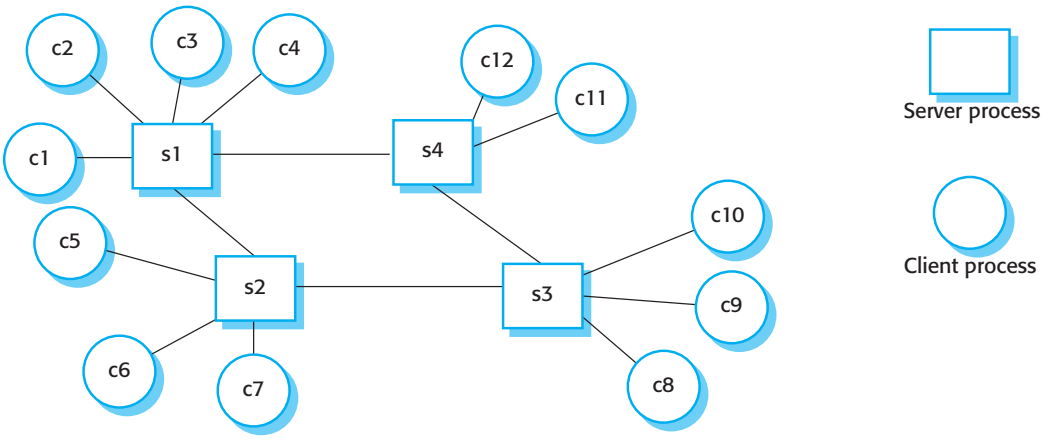


Figure 17.4 Client–server interaction

between these layers. This allows each layer to be distributed to a different computer. Figure 17.6 illustrates this model, showing an application structured into four layers:

1. *A presentation layer* that is concerned with presenting information to the user and managing all user interaction.
2. *A data-handling layer* that manages the data that is passed to and from the client. This layer may implement checks on the data, generate web pages, and so on.
3. *An application processing layer* that is concerned with implementing the logic of the application and so providing the required functionality to end-users.
4. *A database layer* that stores the data and provides transaction management and query services.

The following section explains how different client–server architectures distribute these logical layers in different ways. The client–server model also underlies the notion of software as a service (SaaS), an important way of deploying software and accessing it over the Internet. I cover this topic in Section 17.4.

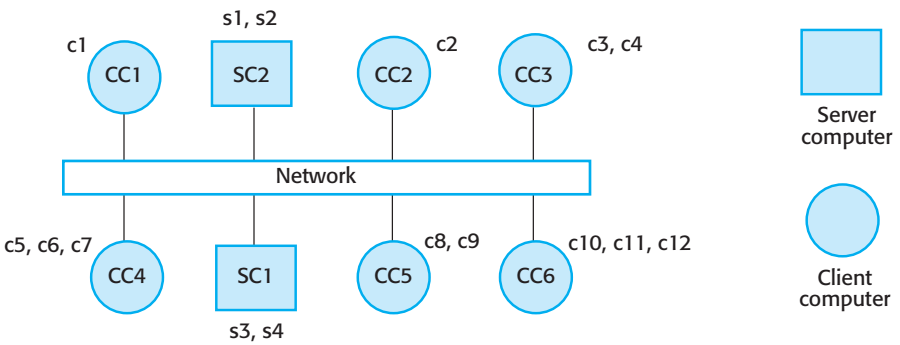


Figure 17.5 Mapping of clients and servers to networked computers

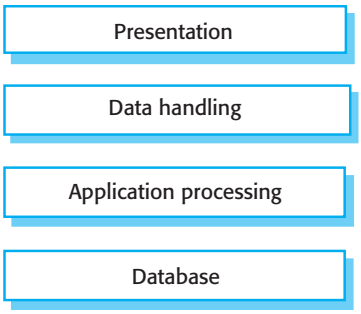


Figure 17.6 Layered architectural model for client–server application

17.3 Architectural patterns for distributed systems

As I explained in the introduction to this chapter, designers of distributed systems have to organize their system designs to find a balance between performance, dependability, security, and manageability of the system. Because no universal model of system organization is appropriate for all circumstances, various distributed architectural styles have emerged. When designing a distributed application, you should choose an architectural style that supports the critical non-functional requirements of your system.

In this section, I discuss five architectural styles:

1. *Master-slave architecture*, which is used in real-time systems in which guaranteed interaction response times are required.
2. *Two-tier client–server architecture*, which is used for simple client–server systems and in situations where it is important to centralize the system for security reasons.
3. *Multi-tier client–server architecture*, which is used when the server has to process a high volume of transactions.
4. *Distributed component architecture*, which is used when resources from different systems and databases need to be combined, or as an implementation model for multi-tier client–server systems.
5. *Peer-to-peer architecture*, which is used when clients exchange locally stored information and the role of the server is to introduce clients to each other. It may also be used when a large number of independent computations may have to be made.

17.3.1 Master–slave architectures

Master–slave architectures for distributed systems are commonly used in real-time systems. In those systems, there may be separate processors associated with data acquisition from the system's environment, data processing and computation,

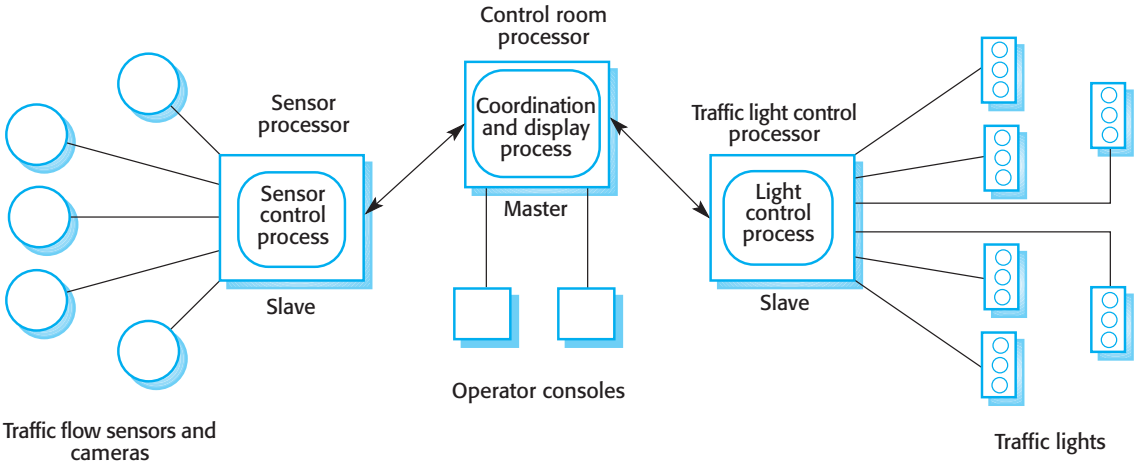


Figure 17.7 A traffic management system with a master–slave architecture

and actuator management. Actuators, as I discuss in Chapter 21, are devices controlled by the software system that act to change the system's environment. For example, an actuator may control a valve and change its state from "open" to "closed." The "master" process is usually responsible for computation, coordination, and communications, and it controls the "slave" processes. "Slave" processes are dedicated to specific actions, such as the acquisition of data from an array of sensors.

Figure 17.7 shows an example of this architectural model. A traffic control system in a city has three logical processes that run on separate processors. The master process is the control room process, which communicates with separate slave processes that are responsible for collecting traffic data and managing the operation of traffic lights.

A set of distributed sensors collects information on the traffic flow. The sensor control process polls the sensors periodically to capture the traffic flow information and collates this information for further processing. The sensor processor is itself polled periodically for information by the master process that is concerned with displaying traffic status to operators, computing traffic light sequences, and accepting operator commands to modify these sequences. The control room system sends commands to a traffic light control process that converts these into signals to control the traffic light hardware. The master control room system is itself organized as a client–server system, with the client processes running on the operator's consoles.

You use this master–slave model of a distributed system in situations where you can predict the distributed processing that is required and where processing can be easily localized to slave processors. This situation is common in real-time systems, where it is important to meet processing deadlines. Slave processors can be used for computationally intensive operations, such as signal processing and the management of equipment controlled by the system.

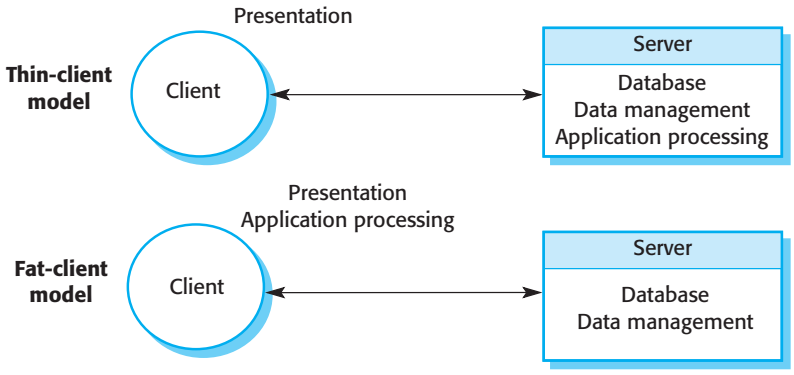


Figure 17.8 Thin- and fat-client architectural models

17.3.2 Two-tier client–server architectures

In Section 17.2, I explained the general organization of client–server systems in which part of the application system runs on the user's computer (the client), and part runs on a remote computer (the server). I also presented a layered application model (Figure 17.6) where the different layers in the system may execute on different computers.

A two-tier client–server architecture is the simplest form of client–server architecture. The system is implemented as a single logical server plus an indefinite number of clients that use that server. This is illustrated in Figure 17.8, which shows two forms of this architectural model:

1. *A thin-client model*, where the presentation layer is implemented on the client and all other layers (data handling, application processing, and database) are implemented on a server. The client presentation software is usually a web browser, but apps for mobile devices may also be available.
2. *A fat-client model*, where some or all of the application processing is carried out on the client. Data management and database functions are implemented on the server. In this case, the client software may be a specially written program that is tightly integrated with the server application.

The advantage of the thin-client model is that it is simple to manage the clients. This becomes a major issue when there are a large number of clients, as it may be difficult and expensive to install new software on all of them. If a web browser is used as the client, there is no need to install any software.

The disadvantage of the thin-client approach, however, is that it places a heavy processing load on both the server and the network. The server is responsible for all computation, which may lead to the generation of significant network traffic between the client and the server. Implementing a system using this model may therefore require additional investment in network and server capacity.

The fat-client model makes use of available processing power on the computer running the client software, and distributes some or all of the application processing

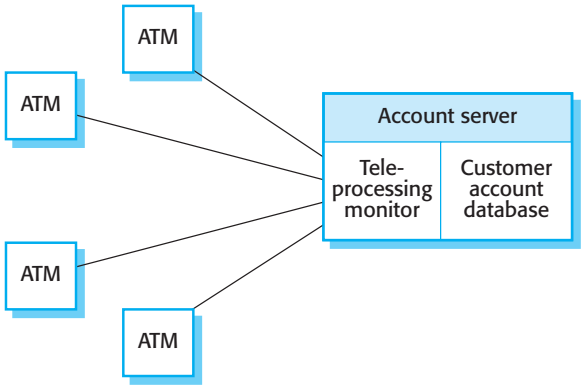


Figure 17.9 A fat-client architecture for an ATM system

and the presentation to the client. The server is essentially a transaction server that manages all database transactions. Data handling is straightforward as there is no need to manage the interaction between the client and the application processing system. The fat-client model requires system management to deploy and maintain the software on the client computer.

An example of a situation in which a fat-client architecture is used is in a bank ATM system, which delivers cash and other banking services to users. The ATM is the client computer, and the server is, typically, a mainframe running the customer account database. A mainframe computer is a powerful machine that is designed for transaction processing. It can therefore handle the large volume of transactions generated by ATMs, other teller systems, and online banking. The software in the teller machine carries out a lot of the customer-related processing associated with a transaction.

Figure 17.9 shows a simplified version of the ATM system organization. The ATMs do not connect directly to the customer database, but rather to a teleprocessing (TP) monitor. A TP monitor is a middleware system that organizes communications with remote clients and serializes client transactions for processing by the database. This ensures that transactions are independent and do not interfere with one other. Using serial transactions means that the system can recover from faults without corrupting the system data.

While a fat-client model distributes processing more effectively than a thin-client model, system management is more complex if a special-purpose client, rather than a browser, is used. Application functionality is spread across many computers. When the application software has to be changed, this involves software reinstallation on every client computer. This can be a major cost if there are hundreds of clients in the system. Auto-update of the client software can reduce these costs but introduces its own problems if the client functionality is changed. The new functionality may mean that businesses have to change the ways they use the system.

The extensive use of mobile devices means that it is important to mimimize network traffic wherever possible. These devices now include powerful computers that can carry out local processing. As a consequence, the distinction between thin-client and fat-client architectures has become blurred. Apps can have inbuilt functionality that carries out local processing, and web pages may include Javascript components

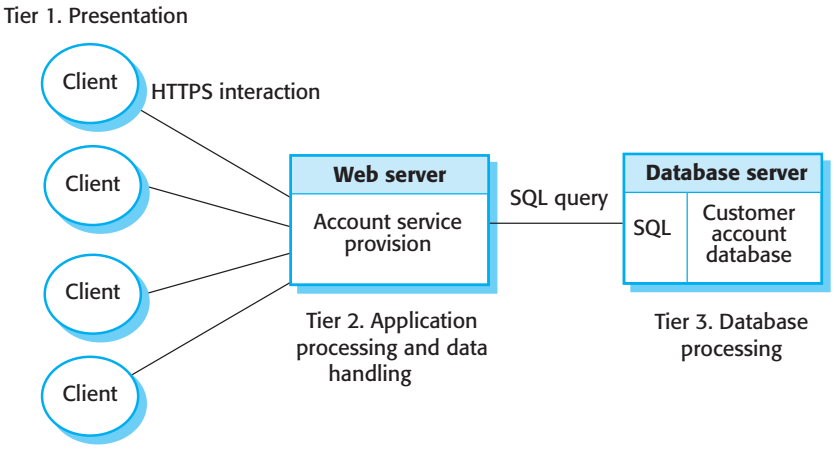


Figure 17.10 Three-tier architecture for an Internet banking system

that execute on the user's local computer. The update problem for apps remains an issue, but it has been addressed, to some extent, by automatically updating apps without explicit user intervention. Consequently, while it is sometimes helpful to use these models as a general basis for the architecture of a distributed system, in practice few web-based applications implement all processing on the remote server.

17.3.3 Multi-tier client–server architectures

The fundamental problem with a two-tier client–server approach is that the logical layers in the system—presentation, application processing, data management, and database—must be mapped onto two computer systems: the client and the server. This may lead to problems with scalability and performance if the thin-client model is chosen, or problems of system management if the fat-client model is used. To avoid some of these problems, a "multi-tier client–server" architecture can be used. In this architecture, the different layers of the system, namely presentation, data management, application processing, and database, are separate processes that may execute on different processors.

An Internet banking system (Figure 17.10) is an example of a multi-tier client–server architecture, where there are three tiers in the system. The bank's customer database (usually hosted on a mainframe computer as discussed above) provides database services. A web server provides data management services such as web page generation and some application services. Application services such as facilities to transfer cash, generate statements, pay bills, and so on are implemented in the web server and as scripts that are executed by the client. The user's own computer with an Internet browser is the client. This system is scalable because it is relatively easy to add servers (scale out) as the number of customers increase.

In this case, the use of a three-tier architecture allows the information transfer between the web server and the database server to be optimized. Efficient middleware that supports database queries in SQL (Structured Query Language) is used to handle information retrieval from the database.

Architecture	Applications
Two-tier client–server architecture with thin clients	Legacy system applications that are used when separating application processing and data handling is impractical. Clients may access these as services, as discussed in Section 17.4. Computationally intensive applications such as compilers with little or no requirements for data handling. Data-intensive applications (browsing and querying) with non-intensive application processing. Simple web browsing is the most common example of a situation where this architecture is used.
Two-tier client–server architecture with fat clients	Applications where application processing is provided by off-the-shelf software (e.g., Microsoft Excel) on the client. Applications where computationally intensive processing of data (e.g., data visualization) is required. Mobile applications where internet connectivity cannot be guaranteed. Local processing using cached information from the database is therefore possible.
Multi-tier client–server architecture	Large-scale applications with hundreds or thousands of clients. Applications where both the data and the application are volatile. Applications where data from multiple sources are integrated.

Figure 17.11 Use of client–server architectural patterns

The three-tier client–server model can be extended to a multi-tier variant, where additional servers are added to the system. This may involve using a web server for data management and separate servers for application processing and database services. Multi-tier systems may also be used when applications need to access and use data from different databases. In this case, you may need to add an integration server to the system. The integration server collects the distributed data and presents it to the application server as if it were from a single database. As I discuss in the following section, distributed component architectures may be used to implement multi-tier client–server systems.

Multi-tier client–server systems that distribute application processing across several servers are more scalable than two-tier architectures. The tiers in the system can be independently managed, with additional servers added as the load increases. Processing may be distributed between the application logic and the data-handling servers, thus leading to more rapid response to client requests.

Designers of client–server architectures must take a number of factors into account when choosing the most appropriate distribution architecture. Situations in which the client–server architectures discussed here may be used are described in Figure 17.11.

17.3.4 Distributed component architectures

By organizing processing into layers, as shown in Figure 17.6, each layer of a system can be implemented as a separate logical server. This model works well for many types of application. However, it limits the flexibility of system designers in that they

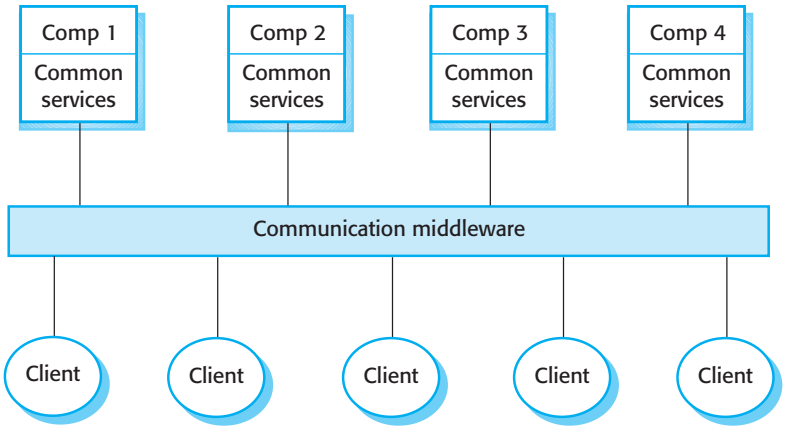


Figure 17.12 A distributed component architecture

have to decide what services should be included in each layer. In practice, however, it is not always clear whether a service is a data management service, an application service, or a database service. Designers must also plan for scalability and so provide some means for servers to be replicated as more clients are added to the system.

A more general approach to distributed system design is to design the system as a set of services, without attempting to allocate these services to layers in the system. Each service, or group of related services, can be implemented using a separate object or component. In a distributed component architecture (Figure 17.12), the system is organized as a set of interacting components as I discussed in Chapter 16. These components provide an interface to a set of services that they provide. Other components call on these services through middleware, using remote procedure or method calls.

Distributed component systems are reliant on middleware. This manages component interactions, reconciles differences between types of the parameters passed between components, and provides a set of common services that application components can use. The CORBA standard (Orfali, Harkey, and Edwards 1997) defined middleware for distributed component systems, but CORBA implementations have never been widely adopted. Enterprises preferred to use proprietary software such as Enterprise Java Beans (EJB) or .NET.

Using a distributed component model for implementing distributed systems has a number of benefits:

1. It allows the system designer to delay decisions on where and how services should be provided. Service-providing components may execute on any node of the network. There is no need to decide in advance whether a service is part of a data management layer, an application layer, or a user interface layer.
2. It is a very open-system architecture that allows new resources to be added as required. New system services can be added easily without major disruption to the existing system.
3. The system is flexible and scalable. New objects or replicated objects can be added as the load on the system increases, without disrupting other parts of the system.

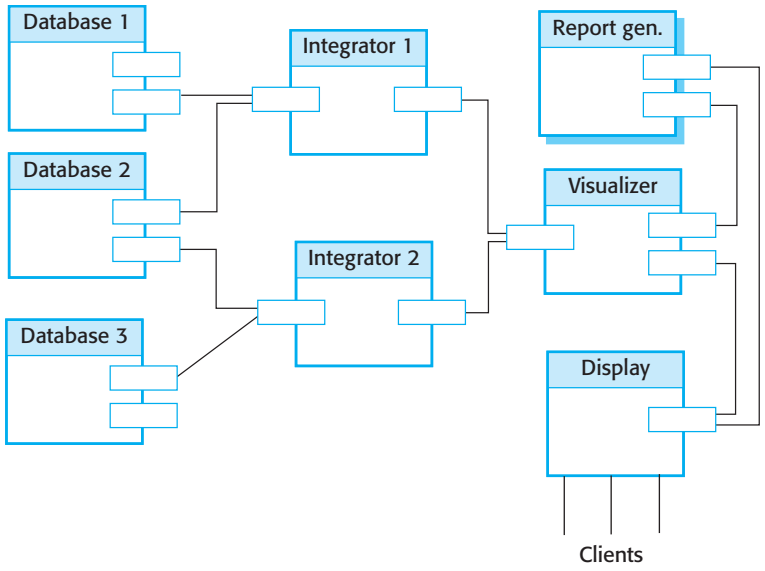


Figure 17.13 A distributed component architecture for a data-mining system

4. It is possible to reconfigure the system dynamically with components migrating across the network as required. This may be important where there are fluctuating patterns of demand on services. A service-providing component can migrate to the same processor as service-requesting objects, thus improving the performance of the system.

A distributed component architecture can be used as a logical model that allows you to structure and organize the system. In this case, you think about how to provide application functionality solely in terms of services and combinations of services. You then work out how to implement these services. For example, a retail application may have application components concerned with stock control, customer communications, goods ordering, and so on.

Data-mining systems are a good example of a type of system that can be implemented using a distributed component architecture. Data-mining systems look for relationships between the data that may be distributed across databases (Figure 17.13). These systems pull in information from several separate databases, carry out computationally intensive processing, and present easy-to-understand visualizations of the relationships that have been discovered.

An example of such a data-mining application might be a system for a retail business that sells food and books. Retail businesses maintain separate databases with detailed information about food products and books. They use a loyalty card system to keep track of customers' purchases, so there is a large database linking bar codes of products with customer information. The marketing department wants to find relationships between a customer's food and book purchases. For instance, a relatively high proportion of people who buy pizzas might also buy crime novels. With this knowledge, the business can specifically target customers who make specific food purchases with information about new novels when they are published.

In this example, each sales database can be encapsulated as a distributed component with an interface that provides read-only access to its data. Integrator components are each concerned with specific types of relationships, and they collect information from all of the databases to try to deduce the relationships. There might be an integrator component that is concerned with seasonal variations in goods sold, and another integrator that is concerned with relationships between different types of goods.

Visualizer components interact with integrator components to create a visualization or a report on the relationships that have been discovered. Because of the large volumes of data that are handled, visualizer components normally present their results graphically. Finally, a display component may be responsible for delivering the graphical models to clients for final presentation in their web browser.

A distributed component architecture rather than a layered architecture is appropriate for this type of application because you can add new databases to the system without major disruption. Each new database is simply accessed by adding another distributed component. The database access components provide a simplified interface that controls access to the data. The databases that are accessed may reside on different machines. The architecture also makes it easy to mine new types of relationships by adding new integrator objects.

Distributed component architectures suffer from two major disadvantages:

1. They are more complex to design than client–server systems. Multilayer client–server systems appear to be a fairly intuitive way to think about systems. They reflect many human transactions where people request and receive services from other people who specialize in providing these services. The complexity of distributed component architectures increases the costs of implementation.
2. There are no universal standards for distributed component models or middleware. Rather, different vendors, such as Microsoft and Sun, developed different, incompatible middleware. This middleware is complex, and reliance on it significantly increases the complexity of distributed component systems.

As a result of these problems, distributed component architectures are being replaced by service-oriented systems (discussed in Chapter 18). However, distributed component systems have performance benefits over service-oriented systems. RPC communications are usually faster than the message-based interaction used in service-oriented systems. Distributed component architectures are therefore still used for high-throughput systems in which large numbers of transactions have to be processed quickly.

17.3.5 Peer-to-peer architectures

The client–server model of computing that I have discussed in previous sections of the chapter makes a clear distinction between servers, which are providers of services, and clients, which are receivers of services. This model usually leads to an uneven distribution of load on the system, where servers do more work than clients. This may lead to organizations spending a lot on server capacity while there is unused processing capacity on the hundreds or thousands of PCs and mobile devices used to access the system servers.

Peer-to-peer (p2p) systems (Oram 2001) are decentralized systems in which computations may be carried out by any node on the network. In principle at least, no distinctions are made between clients and servers. In peer-to-peer applications, the overall system is designed to take advantage of the computational power and storage available across a potentially huge network of computers. The standards and protocols that enable communications across the nodes are embedded in the application itself, and each node must run a copy of that application.

Peer-to-peer technologies have mostly been used for personal rather than business systems. The fact that there are no central servers means that these systems are harder to monitor; therefore, a higher level of communication privacy is possible.

For example, file-sharing systems based on the BitTorrent protocol are widely used to exchange files on users' PCs. Private instant messaging systems, such as ICQ and Jabber, provide direct communications between users without an intermediate server. Bitcoin is a peer-to-peer payments system using the Bitcoin electronic currency. Freenet is a decentralized database that has been designed to make it easier to publish information anonymously and to make it difficult for authorities to suppress this information.

Other p2p systems have been developed where privacy is not the principal requirement. Voice over IP (VoIP) phone services, such as Viber, rely on peer-to-peer communication between the parties involved in the phone call or conference. SETI@home is a long-running project that processes data from radio telescopes on home PCs in order to search for indications of extraterrestrial life. In these systems, the advantage of the p2p model is that a central server is not a processing bottleneck.

Peer-to-peer systems have also been used by businesses to harness the power in their PC networks (McDougall 2000). Intel and Boeing have both implemented p2p systems for computationally intensive applications. Such systems take advantage of unused processing capacity on local computers. Instead of buying expensive high-performance hardware, engineering computations can be run overnight when desktop computers are unused. Businesses also make extensive use of commercial p2p systems, such as messaging and VoIP systems.

In principle, every node in a p2p network could be aware of every other node. Nodes could connect to and exchange data directly with any other node in the network. In practice, this is impossible unless the network has only a few members. Consequently, nodes are usually organized into "localities," with some nodes acting as bridges to other node localities. Figure 17.14 shows this decentralized p2p architecture.

In a decentralized architecture, the nodes in the network are not simply functional elements but are also communications switches that can route data and control signals from one node to another. For example, assume that Figure 17.14 represents a decentralized, document-management system. A consortium of researchers uses this system to share documents. Each member of the consortium maintains his or her own document store. However, when a document is retrieved, the node retrieving that document also makes it available to other nodes.

If someone needs a document that is stored somewhere on the network, they issue a search command, which is sent to nodes in their "locality." These nodes check whether they have the document and, if so, return it to the requestor. If they do not have it, they route the search to other nodes. Therefore if n1 issues a search for a

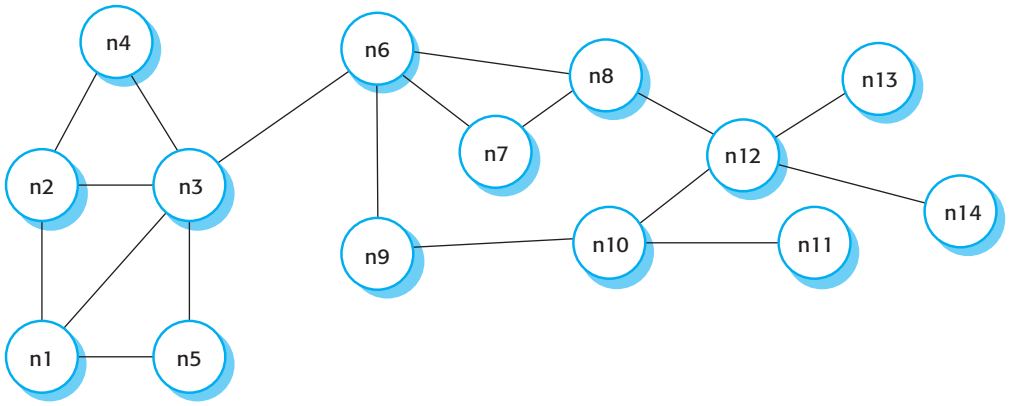


Figure 17.14
A decentralized p2p architecture

document that is stored at n10, this search is routed through nodes n3, n6, and n9 to n10. When the document is finally discovered, the node holding the document then sends it to the requesting node directly by making a peer-to-peer connection.

This decentralized architecture has the advantage of being highly redundant and hence both fault-tolerant and tolerant of nodes disconnecting from the network. However, the disadvantages are that many different nodes may process the same search, and there is also significant overhead in replicated peer communications.

An alternative p2p architectural model, which departs from a pure p2p architecture, is a semicentralized architecture where, within the network, one or more nodes act as servers to facilitate node communications. This reduces the amount of traffic between nodes. Figure 17.15 illustrates how this semicentralized architectural model differs from the completely decentralized model shown in Figure 17.14.

In a semicentralized architecture, the role of the server (sometimes called a superpeer) is to help establish contact between peers in the network or to coordinate the results of a computation. For example, if Figure 17.15 represents an instant messaging system, then network nodes communicate with the server (indicated by dashed lines) to find out what other nodes are available. Once these nodes are discovered, direct communications can be established and the connection to the server becomes unnecessary. Therefore, nodes n2, n3, n5, and n6 are in direct communication.

In a computational p2p system, where a processor-intensive computation is distributed across a large number of nodes, it is normal for some nodes to be superpeers. Their role is to distribute work to other nodes and to collate and check the results of the computation.

The peer-to-peer architectural model may be the best model for a distributed system in two circumstances:

1. Where the system is computationally-intensive and it is possible to separate the processing required into a large number of independent computations. For example, a peer-to-peer system that supports computational drug discovery distributes computations that look for potential cancer treatments by analyzing a huge number of molecules to see if they have the characteristics required to suppress the growth of cancers. Each molecule can be considered separately, so there is no need for the peers in the system to communicate.

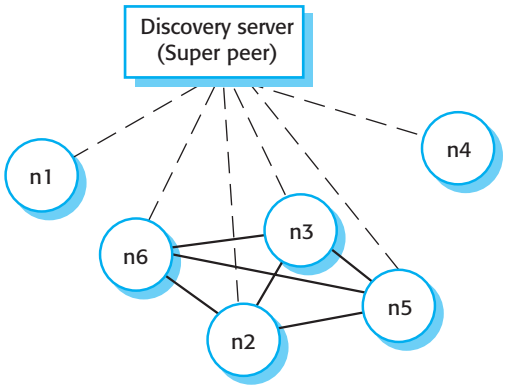


Figure 17.15 A semicentralized p2p architecture

2. Where the system primarily involves the exchange of information between individual computers on a network and there is no need for this information to be centrally stored or managed. Examples of such applications include file-sharing systems that allow peers to exchange local files such as music and video files, and phone systems that support voice and video communications between computers.

Peer-to-peer architectures allow for the efficient use of capacity across a network. However, security concerns are the principal reason why these systems have not become more widely used, especially in business (Wallach 2003). The lack of centralized management means that attackers can set up malicious nodes that deliver spam and malware to legitimate p2p system users. Peer-to-peer communications involve opening your computer to direct interactions with other peers and this means that these systems could potentially access any of your resources. To counter this possibility, you need to organize your system so that these resources are protected. If this is done incorrectly, then your system is insecure and vulnerable to external corruption.

17.4 Software as a service

In the previous sections, I discussed client–server models and how functionality may be distributed between the client and the server. To implement a client–server system, you may have to install a program or an app on the client computer, which communicates with the server, implements client-side functionality, and manages the user interface. For example, a mail client, such as Outlook or Mac Mail, provides mail management features on your own computer. This avoids the problem of server overload in thin-client systems, where all of the processing is carried out at the server.

The problems of server overload can be significantly reduced by using web technologies such as AJAX (Holdener, 2008) and HTML5 (Sarris 2013). These technologies support efficient management of web page presentation and local computation by executing scripts that are part of the web page. This means that a browser can be configured and used as client, with significant local processing. The application software can be

thought of as a remote service, which can be accessed from any device that can run a standard browser. Widely used examples of SaaS include web-based mail systems, such as Yahoo and Gmail, and office applications, such as Google Docs and Office 365.

This idea of software as a service (SaaS) involves hosting the software remotely and providing access to it over the Internet. The key elements of SaaS are as follows:

1. Software is deployed on a server (or more commonly in the cloud) and is accessed through a web browser. It is not deployed on a local PC.
2. The software is owned and managed by a software provider rather than the organizations using the software.
3. Users may pay for the software according to how much use they make of it or through an annual or monthly subscription. Sometimes the software is free for anyone to use, but users must then agree to accept advertisements, which fund the software service.

The development of SaaS has accelerated over the past few years as cloud computing has become widely used. When a service is deployed in the cloud, the number of servers can quickly change to match the user demands for that service. There is no need for service providers to provision for peak loads; as a result, the costs for these providers have been dramatically reduced.

For software purchasers, the benefit of SaaS is that the costs of management of software are transferred to the provider. The provider is responsible for fixing bugs and installing software upgrades, dealing with changes to the operating system platform, and ensuring that hardware capacity can meet demand. Software license management costs are zero. If someone has several computers, there is no need to license software for all of these. If a software application is only used occasionally, the pay-per-use model may be cheaper than buying an application. The software may be accessed from mobile devices, such as smartphones, from anywhere in the world.

The main problem that inhibits the use of SaaS is data transfer with the remote service. Data transfer takes place at network speeds, and so transferring a large amount of data, such as video or high-quality images takes a lot of time. You may also have to pay the service provider according to the amount transferred. Other problems are lack of control over software evolution (the provider may change the software when it wishes) and problems with laws and regulations. Many countries have laws governing the storage, management, preservation, and accessibility of data, and moving data to a remote service may breach these laws.

The notion of software as a service and service-oriented architectures (SOA), discussed in Chapter 18, are related, but they are not the same:

1. Software as a service is a way of providing functionality on a remote server with client access through a web browser. The server maintains the user's data and state during an interaction session. Transactions are usually long transactions, for example, editing a document.

2. Service-oriented architecture is an approach to structuring a software system as a set of separate, stateless services. These services may be provided by multiple providers and may be distributed. Typically, transactions are short transactions where a service is called, does something, and then returns a result.

SaaS is a way of delivering application functionality to users, whereas SOA is an implementation technology for application systems. Systems that are implemented using SOA do not have to be accessed by users as web services. SaaS applications for business may be implemented using components rather than services. However, if SaaS is implemented using SOA, it becomes possible for applications to use service APIs to access the functionality of other applications. They can then be integrated into more complex systems. These systems are called mashups and are another approach to software reuse and rapid software development.

From a software development perspective, the process of service development has much in common with other types of software development. However, service construction is not usually driven by user requirements, but by the service provider's assumptions about what users need. Accordingly, the software needs to be able to evolve quickly after the provider gets feedback from users on their requirements. Agile development with incremental delivery is therefore an effective approach for software that is to be deployed as a service.

Some software that is implemented as a service, such as Google Docs for web users, offers a generic experience to all users. However, businesses may wish to have specific services that are tailored to their own requirements. If you are implementing SaaS for business, you may base your software service on a generic service that is tailored to the needs of each business customer. Three important factors have to be considered:

1. *Configurability* How do you configure the software for the specific requirements of each organization?
2. *Multi-tenancy* How do you present each user of the software with the impression that they are working with their own copy of the system while, at the same time, making efficient use of system resources?
3. *Scalability* How do you design the system so that it can be scaled to accommodate an unpredictably large number of users?

The notion of product-line architectures, discussed in Chapter 16, is one way of configuring software for users who have overlapping but not identical requirements. You start with a generic system and adapt it according to the specific requirements of each user.

This does not work for SaaS, however, for it would mean deploying a different copy of the service for each organization that uses the software. Rather, you need to design configurability into the system and provide a configuration interface that allows users to specify their preferences. You then use these preferences to adjust the behavior of the software dynamically as it is used. Configuration facilities may allow for:

1. *Branding*, where users from each organization are presented with an interface that reflects their own organization.

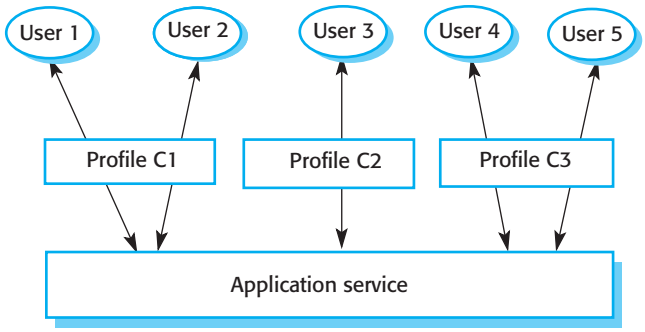


Figure 17.16 Configuration of a software system offered as a service

2. *Business rules and workflows*, where each organization defines its own rules that govern the use of the service and its data.
3. *Database extensions*, where each organization defines how the generic service data model is extended to meet its specific needs.
4. *Access control*, where service customers create individual accounts for their staff and define the resources and functions that are accessible to each of their users.

Figure 17.16 illustrates this situation. This diagram shows five users of the application service, who work for three different customers of the service provider. Users interact with the service through a customer profile that defines the service configuration for their employer.

Multi-tenancy is a situation in which many different users access the same system and the system architecture is defined to allow the efficient sharing of system resources. However, it must appear to users that they each have sole use of the system. Multi-tenancy involves designing the system so that there is an absolute separation between system functionality and system data. All operations must therefore be stateless so that they can be shared. Data must either be provided by the client or should be available in a storage system or database that can be accessed from any system instance.

A particular problem in multi-tenant systems is data management. The simplest way to provide data management is for all customers to have their own database, which they may use and configure as they wish. However, this requires the service provider to maintain many different database instances (one per customer) and to make these databases available on demand.

As an alternative, the service provider can use a single database, with different users being virtually isolated within that database. This is illustrated in Figure 17.17, where you can see that database entries also have a "tenant identifier" that links these entries to specific users. By using database views, you can extract the entries for each service customer and so present users from that customer with a virtual, personal database. This process can be extended to meet specific customer needs using the configuration features discussed above.

Scalability is the ability of the system to cope with increasing numbers of users without reducing the overall quality of service that is delivered to any user. Generally,

Tenant	Key	Name	Address
234	C100	XYZ Corp	43, Anystreet, Sometown
234	C110	BigCorp	2, Main St, Motown
435	X234	J. Bowie	56, Mill St, Starville
592	PP37	R. Burns	Alloway, Ayrshire

Figure 17.17 A multi-tenant database

when considering scalability in the context of SaaS, you are considering "scaling out" rather than "scaling up." Recall that scaling out means adding additional servers and so also increasing the number of transactions that can be processed in parallel. Scalability is a complex topic that I cannot cover in detail here, but following are some general guidelines for implementing scalable software:

1. Develop applications where each component is implemented as a simple stateless service that may be run on any server. In the course of a single transaction, a user may therefore interact with instances of the same service that are running on several different servers.
2. Design the system using asynchronous interaction so that the application does not have to wait for the result of an interaction (such as a read request). This allows the application to carry on doing useful work while it is waiting for the interaction to finish.
3. Manage resources, such as network and database connections, as a pool so that no single server is likely to run out of resources.
4. Design your database to allow fine-grain locking. That is, do not lock out whole records in the database when only part of a record is in use.
5. Use a cloud PaaS platform, such as Google App Engine (Sanderson 2012) or other PaaS platform for system implementation. These include mechanisms that will automatically scale out your system as the load increases.

The notion of software as a service is a major paradigm shift for distributed computing. We have already seen consumer software and professional applications, such as Photoshop, move to this model of delivery. Increasingly, businesses are replacing their own systems, such as CRM and inventory systems, with cloud-based SaaS systems from external providers such as Salesforce. Specialized software companies that implement business applications prefer to provide SaaS because it simplifies software update and management.

SaaS represents a new way to think about the engineering of enterprise systems. It has always been helpful to think of systems delivering services to users, but, until SaaS, this function has involved using different abstractions, such as objects, when implementing the system. Where there is a closer match between user and system abstractions, the resultant systems are easier to understand, maintain, and evolve.

KEY POINTS

- The benefits of distributed systems are that they can be scaled to cope with increasing demand, can continue to provide user services (even if some parts of the system fail), and they enable resources to be shared.
- Issues to be considered in the design of distributed systems include transparency, openness, scalability, security, quality of service, and failure management.
- Client–server systems are distributed systems in which the system is structured into layers, with the presentation layer implemented on a client computer. Servers provide data management, application, and database services.
- Client–server systems may have several tiers, with different layers of the system distributed to different computers.
- Architectural patterns for distributed systems include master–slave architectures, two-tier and multi-tier client–server architectures, distributed component architectures, and peer-to-peer architectures.
- Distributed component systems require middleware to handle component communications and to allow objects to be added to and removed from the system.
- Peer-to-peer architectures are decentralized architectures in which there are no distinguished clients and servers. Computations can be distributed over many systems in different organizations.
- Software as a service is a way of deploying applications as thin client–server systems, where the client is a web browser.

FURTHER READING

Peer-to-Peer: Harnessing the Power of Disruptive Technologies. Although this book does not have a lot of information on p2p architectures, it is an excellent introduction to p2p computing and discusses the organization and approach used in a number of p2p systems. (A. Oram (ed.), O'Reilly and Associates Inc., 2001).

"Turning Software into a Service." A good overview paper that discusses the principles of service-oriented computing. Unlike many papers on this topic, it does not conceal these principles behind a discussion of the standards involved. (M. Turner, D. Budgen, and P. Brereton, *IEEE Computer*, 36 (10), October 2003) http://dx.doi.org/10.1109/MC.2003.1236470

Distributed Systems, 5th ed. A comprehensive textbook that discusses all aspects of distributed systems design and implementation. It includes coverage of peer-to-peer systems and mobile systems. (G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. Addison-Wesley, 2011).

Engineering Software as a Service: An Agile Approach Using Cloud Computing. This book accompanies the authors' online course on the topic. A good practical book that is aimed at people new to this type of development. (A. Fox and D. Patterson, Strawberry Canyon LLC, 2014) http://www.saasbook.info

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/requirements-and-design/

EXERCISES

17.1. What do you understand by "scalability"? Discuss the differences between scaling up and scaling out and explain when these different approaches to scalability may be used.

17.2. Explain why distributed software systems are more complex than centralized software systems, where all of the system functionality is implemented on a single computer.

17.3. Using an example of a remote procedure call, explain how middleware coordinates the interaction of computers in a distributed system.

17.4. What are the different logical layers in an application with a distributed client–server architecture?

17.5. You have been asked to design a secure system that requires strong authentication and authorization. The system must be designed so that communications between parts of the system cannot be intercepted and read by an attacker. Suggest the most appropriate client–server architecture for this system and, giving the reasons for your answer, propose how functionality should be distributed between the client and the server systems.

17.6. Your customer wants to develop a system for stock information where dealers can access information about companies and evaluate various investment scenarios using a simulation system. Each dealer uses this simulation in a different way, according to his or her experience and the type of stocks in question. Suggest a client–server architecture for this system that shows where functionality is located. Justify the client–server system model that you have chosen.

17.7. Using a distributed component approach, propose an architecture for a national theater booking system. Users can check seat availability and book seats at a group of theaters. The system should support ticket returns so that people may return their tickets for last-minute resale to other customers.

17.8. What is the fundamental problem with a two-tier client–server approach? Define how a multi-tier client–server approach overcomes this.

17.9. List the benefits that a distributed component model has when used for implementing distributed systems.

17.10. Your company wishes to move from using desktop applications to accessing the same functionality remotely as services. Identify three risks that might arise and suggest how these risks may be reduced.

REFERENCES

Bernstein, P. A. 1996. "Middleware: A Model for Distributed System Services." *Comm. ACM* 39 (2): 86–97. doi:10.1145/230798.230809.

Coulouris, G., J. Dollimore, T. Kindberg, and G. Blair. 2011. *Distributed Systems: Concepts and Design, 5th ed.* Harlow, UK: Addison-Wesley.

Holdener, A. T. (2008). *Ajax: The Definitive Guide. Sebastopol, CA.: O' Reilly & Associates*.

McDougall, P. 2000. "The Power of Peer-to-Peer." *Information Week* (August 28, 2000). http://www.informationweek.com/801/peer.htm

Oram, A. 2001. "Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology." Sebastopol, CA: O'Reilly & Associates.

Orfali, R., D. Harkey, and J. Edwards. 1997. *Instant CORBA*. Chichester, UK: John Wiley & Sons.

Pope, A. 1997. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Harlow, UK: Addison-Wesley.

Sanderson, D. 2012. *Programming with Google App Engine*. Sebastopol, CA: O'Reilly Media Inc.

Sarris, S. 2013. *HTML5 Unleashed*. Indianapolis, IN: Sams Publishing.

Tanenbaum, A. S., and M. Van Steen. 2007. *Distributed Systems: Principles and Paradigms, 2nd ed.* Upper Saddle River, NJ: Prentice-Hall.

Wallach, D. S. 2003. "A Survey of Peer-to-Peer Security Issues." In *Software Security: Theories and Systems*, edited by M. Okada, B. C. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, 42–57. Heidelberg: Springer-Verlag. doi:10.1007/3-540-36532-X_4.