



16

Software reuse

Objectives

The objectives of this chapter are to introduce software reuse and to describe approaches to system development based on large-scale system 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 configurable, reusable components;
- have learned how systems can be developed by configuring and composing off-the-shelf application software systems.

Contents

- 16.1** The reuse landscape
- 16.2** Application frameworks
- 16.3** Software product lines
- 16.4** COTS product reuse

Reuse-based software engineering is a software engineering strategy where the development process is geared to reusing existing software. Although reuse was proposed as a development strategy more than 40 years ago (McIlroy, 1968), it is only since 2000 that ‘development with reuse’ has become the norm for 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. More and more companies see their software as a valuable asset. They are promoting reuse to increase their return on software investments.

The availability of reusable software has increased dramatically. The open source movement has meant that there is a huge reusable code base available at low cost. This may be in the form of program libraries or entire applications. There are many domain-specific application systems available that can be tailored and adapted to the needs of a specific company. Some large companies provide a range of reusable components for their customers. Standards, such as web service standards, have made it easier to develop general 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. *Application system reuse* The whole of an application system may be reused by incorporating it without changing into other systems or by configuring the application for different customers. Alternatively, application families that have a common architecture, but which are tailored for specific customers, may be developed. I cover application system reuse later in this chapter.
2. *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. I cover component reuse in Chapters 17 and 19.
3. *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, based 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 expertise is needed to develop efficient objects and functions, this is a particularly effective approach.

Software systems and components are potentially reusable entities, but their specific nature sometimes means that it is expensive to modify them for a new situation. A complementary form of reuse is ‘concept reuse’ where, rather than reuse a software component, you reuse an idea, a way, or working or an algorithm. The concept that you reuse is represented in an abstract notation (e.g., a system model), which does not include implementation detail. It can, therefore, be configured and adapted for a range of situations. Concept reuse can be embodied in approaches such as design

Benefit	Explanation
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.
Reduced process risk	The cost of existing software is already known, whereas 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 particularly true when relatively large software components such as subsystems are reused.
Effective use of specialists	Instead of doing the same work over and over again, application specialists can develop reusable software that encapsulates their knowledge.
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.
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.

Figure 16.1 Benefits of software reuse

patterns (covered in Chapter 7), configurable system products, and program generators. When concepts are reused, the reuse process includes an activity where the abstract concepts are instantiated to create executable reusable components.

An obvious advantage of software reuse is that overall development costs should be reduced. Fewer software components need to be specified, designed, implemented, and validated. However, cost reduction is only one advantage of reuse. In Figure 16.1, I have listed other advantages of reusing software assets.

However, there are costs and problems associated with reuse (Figure 16.2). 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 reductions in overall development costs through reuse may be less than anticipated.

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.

Software reuse is most effective when it is planned as part of an organization-wide reuse program. A reuse program involves the creation of reusable assets and the adaptation of development processes to incorporate these assets in new software. The importance of reuse planning has been recognized for many years in Japan (Matsumoto, 1984), where reuse is an integral part of the Japanese ‘factory’ approach

Problem	Explanation
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 increasingly incompatible with system changes.
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 particularly true for tools that support embedded systems engineering, less so 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.
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.

Figure 16.2 Problems with reuse

to software development (Cusamano, 1989). Companies such as Hewlett-Packard have also been very successful in their reuse programs (Griss and Wosser, 1995), and their experience has been documented in a book by Jacobson et al. (1997).

16.1 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 16.3 sets out a number of possible ways of implementing software reuse, with each described briefly in Figure 16.4.

Given this array of techniques for reuse, the key question is “which is the most appropriate technique to use in a particular situation?” Obviously, this depends on the requirements for the system being developed, the technology 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 off-the-shelf systems rather than individual components. These are large-grain reusable assets. Although the fit to

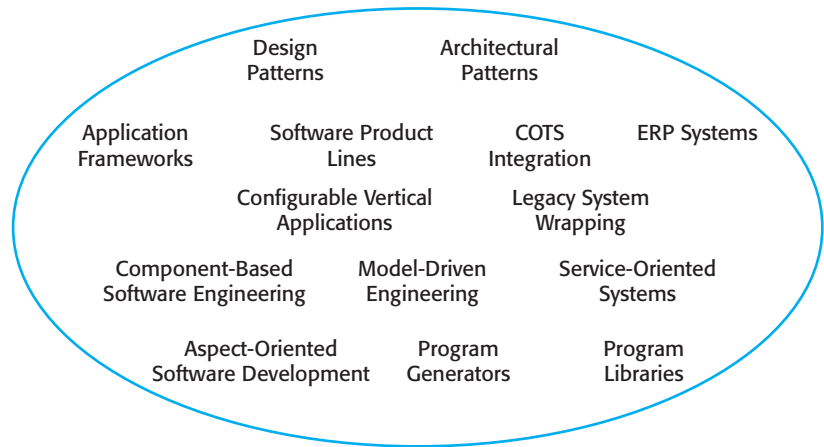


Figure 16.3 The reuse landscape

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, you may prefer to avoid off-the-shelf components and systems from external suppliers; suppliers may not be able to continue support for the reused software.

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, if the development team has skills in a particular area, this is probably where you should focus.
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 dependability case for the system (discussed in Chapter 15). 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 generator-based reuse, where you generate the code from a reusable domain-specific representation of a system. These systems often generate relatively inefficient code.
5. *The application domain* In some application domains, such as manufacturing and medical information systems, there are several generic products that may be reused by configuring them to a local situation. If you are working in such a domain, you should always consider these as an option.

Approach	Description
Architectural patterns	Standard software architectures that support common types of application systems are used as the basis of applications. Described in Chapters 6, 13, and Chapter 20.
Design patterns	Generic abstractions that occur across applications are represented as design patterns showing abstract and concrete objects and interactions. Described in Chapter 7.
Component-based development	Systems are developed by integrating components (collections of objects) that conform to component-model standards. Described in Chapter 17.
Application frameworks	Collections of abstract and concrete classes are adapted and extended to create application systems.
Legacy system wrapping	Legacy systems (see Chapter 9) are ‘wrapped’ by defining a set of interfaces and providing access to these legacy systems through these interfaces.
Service-oriented systems	Systems are developed by linking shared services, which may be externally provided. Described in Chapter 19.
Software product lines	An application type is generalized around a common architecture so that it can be adapted for different customers.
COTS product reuse	Systems are developed by configuring and integrating existing application systems.
ERP systems	Large-scale systems that encapsulate generic business functionality and rules are configured for an organization.
Configurable vertical applications	Generic systems are designed so that they can be configured to the needs of specific system customers.
Program libraries	Class and function libraries that implement commonly used abstractions are available for reuse.
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.
Aspect-oriented software development	Shared components are woven into an application at different places when the program is compiled. Described in Chapter 21.

Figure 16.4
Approaches that
support software
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.



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://www.SoftwareEngineering-9.com/Web/Reuse/Generator.html>

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 to unknown risks.

16.2 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 small and are specialized for a particular application. It 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. (2004) define a framework to be:

“... an integrated set of software artefacts (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 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

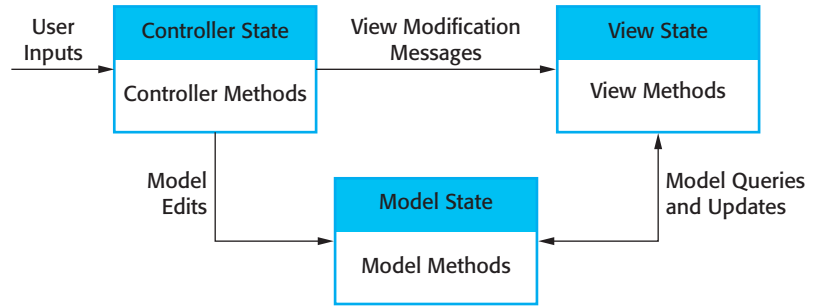


Figure 16.5 The Model-View-Controller pattern

is defined by the object classes and their interactions. Classes are reused directly and may be extended using features such as inheritance.

Frameworks are implemented as a collection of concrete and abstract object classes in an object-oriented programming language. Therefore, frameworks are language-specific. There are frameworks available in all of the commonly used object-oriented programming languages (e.g., Java, C#, C++, as well as dynamic languages such as Ruby and Python). In fact, a framework can incorporate several other frameworks, where each of these 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.

Fayad and Schmidt (1997) discuss three classes of frameworks:

1. *System infrastructure frameworks* These frameworks support the development of system infrastructures such as communications, user interfaces, and compilers (Schmidt, 1997).
2. *Middleware integration frameworks* These 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 17.
3. *Enterprise application frameworks* These 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.

Web application frameworks (WAFs) are a more recent and very important type of framework. WAFs that support the construction of dynamic websites are now widely available. The architecture of a WAF is usually based on the Model-View-Controller (MVC) composite pattern (Gamma et al., 1995), shown in Figure 16.5.

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. It allows for the separation of the application

state from the user interface to application. 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.

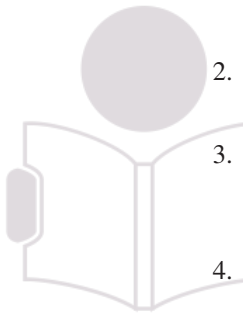
Frameworks are often implementations of design patterns, as discussed in Chapter 7. For example, an MVC framework includes the Observer pattern, the Strategy pattern, the Composite pattern, and a number of others that are discussed by Gamma et al. (1995). The general nature of patterns and their use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

Web application frameworks usually incorporate one or more specialized frameworks that support specific application features. Although each framework includes slightly different functionality, most web application frameworks support the following features:

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 support* Frameworks don't usually include a database but rather assume that a separate database, such as MySQL, will be used. The framework may provide 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* Most web frameworks now provide AJAX support (Holdener, 2008), which allows more interactive web pages to be created.

To extend a framework you do not change the framework code. Rather, you add concrete classes that inherit operations from abstract classes in the framework. In addition, you may have to define callbacks. Callbacks are methods that are called in response to events recognized by the framework. Schmidt et al. (2004) call this 'inversion of control'. The framework objects, rather than the application-specific objects, are responsible for control in the system. In response to events from the user interface, database, etc., these framework objects invoke 'hook methods' that are then linked to user-provided functionality. The application-specific functionality responds to the event in an appropriate way (Figure 16.6). For example, a framework will have a method that handles a mouse click from the environment. This method calls the hook method, which you must configure to call the appropriate application methods to handle the mouse click.

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



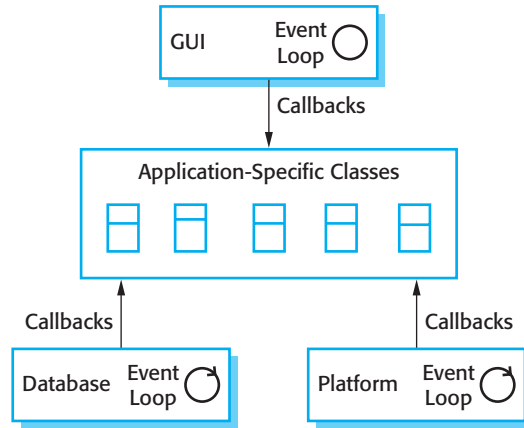


Figure 16.6 Inversion of control in frameworks

create instances of the system is often a straightforward process. It involves rewriting concrete classes and methods that you have added to the framework.

However, frameworks are usually more general than software product lines, which focus on a specific family of application system. For example, you can use a web-based framework to build different types of web-based applications. One of these might be 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.

Frameworks are an effective approach to reuse, but are expensive to introduce into software development processes. 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 difficult because you may not understand how the framework methods interact. This is a general problem with reusable software. Debugging tools may provide information about the reused system components, which a developer does not understand.

16.3 Software product lines

One of the most effective approaches to reuse is to create software product lines or application families. A software product line is a set of applications with a common architecture and shared components, with each application specialized to reflect different requirements. The core system is designed to be configured and adapted to suit the needs of different system customers. 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. Furthermore, application experience is often transferable from one system to another. Consequently, when software engineers join a development team, their learning process is shortened. Testing is simplified because tests for large parts of the application may also be reused, thus reducing the overall application development time.

Software product lines usually emerge from existing applications. That is, an organization develops an application 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 including this in a base application, which is then used for future development. This base application is deliberately structured to simplify reuse and reconfiguration.

Application frameworks and software product lines obviously 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:

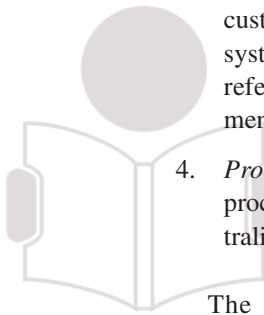
1. Application frameworks rely on object-oriented features such as inheritance and polymorphism to implement extensions to the framework. Generally, the framework code is not modified and the possible modifications are limited to whatever is allowed by the framework. Software product lines are not necessarily 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. Application frameworks are primarily focused on providing technical 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 rarely provide support for hardware interfacing.
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.

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

1. *Platform specialization* Versions of the application are 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 are created to handle particular operating environments and peripheral devices. For example, a system for the emergency services may exist in different versions, depending on the vehicle communications system. In this case, the system components are changed to reflect the functionality of the communications equipment used.
3. *Functional specialization* Versions of the application are 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 is 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 a distributed process in another.



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 despatching 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 this for police, fire, and ambulance services.

This vehicle despatching system is an example of a resource management architecture (Figure 16.7). You can see how this four-layer structure is instantiated in Figure 16.8, which shows the modules that might be included in a vehicle despatching system product line. The components at each level in the product line system are as follows:

1. At the interaction level, there are components providing an operator display interface and an interface with the communications systems used.
2. At the I/O management level (level 2), there are components that handle operator authentication, generate reports of incidents and vehicles despatched, support map output and route planning, and provide a mechanism for operators to query the system databases.

Figure 16.7 The architecture of a resource allocation system

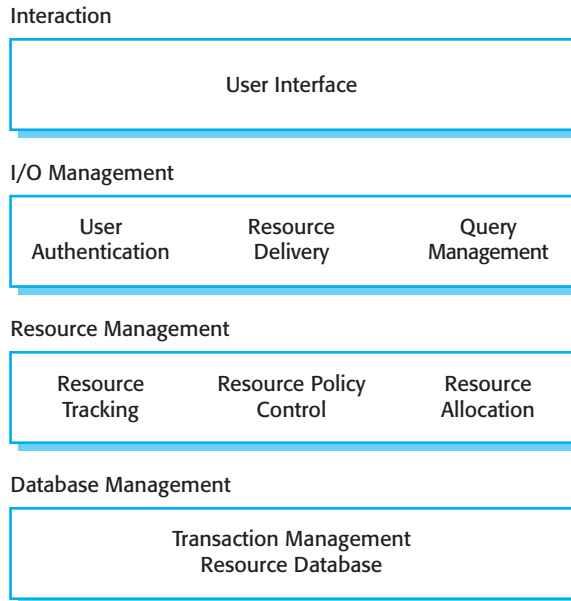
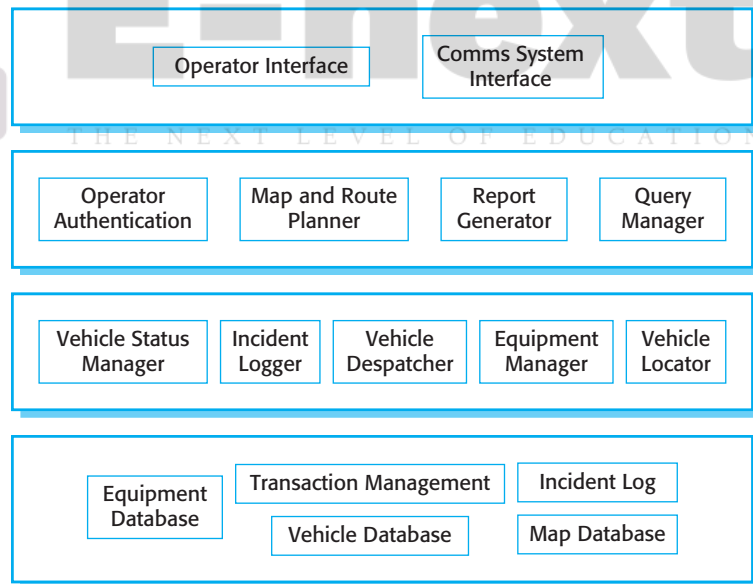


Figure 16.8 The product line architecture of a vehicle dispatcher system



3. At the resource management level (level 3) there are components that allow vehicles to be located and despatched, components to update the status of vehicles and equipment, and a component to 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.

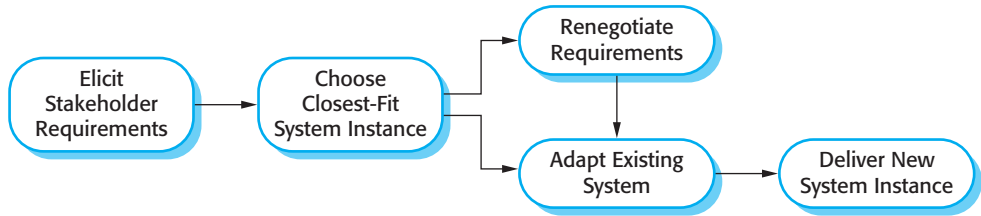


Figure 16.9 Product instance development

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

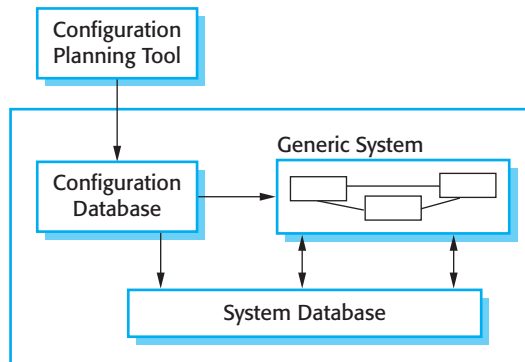
Figure 16.9 shows the steps involved in extending a software product line to create a new application. The steps involved in this general process are as follows:

1. *Elicit stakeholder requirements* You may start with a normal requirements engineering process. However, because a system already exists, you will need to 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, there may be some requirements renegotiation to minimize the changes that are needed.
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 family member* The new instance of the product line is delivered to the customer. 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 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 reconfigured and this reconfiguration may involve adding or removing components from the system, defining parameters and constraints for system components, and including knowledge of business

Figure 16.10
Deployment-time
configuration



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 a set of configuration files that are 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, they create a specific system that delivers the required customer functionality. This usually involves changing and extending the source code of the system so greater flexibility is possible than with deployment-time configuration.

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 16.10). The executing system consults this database when executing so that its functionality may be specialized to its execution context.

There are several levels of deployment-time configuration that 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 it may take many months to configure the system for a customer. Large configurable systems may support the configuration process by providing software tools, such as a configuration planning tools, to support the configuration process. I discuss deployment-time configuration further in Section 16.4.1. This covers the reuse of COTS systems that have to be configured to work in different operational environments.

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.

16.4 COTS product reuse

A commercial-off-the-shelf (COTS) product is a software system that can be adapted to the needs of different customers without changing the source code of the system. Virtually all desktop software and a wide variety of server products are COTS software. Because this software is designed for general use, it usually 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 (2004) also discovered that using open source products were often used as COTS products. That is, the open source systems were used without change and without looking at the source code.

COTS 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 patient. 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 over the last 15 or so years, 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.

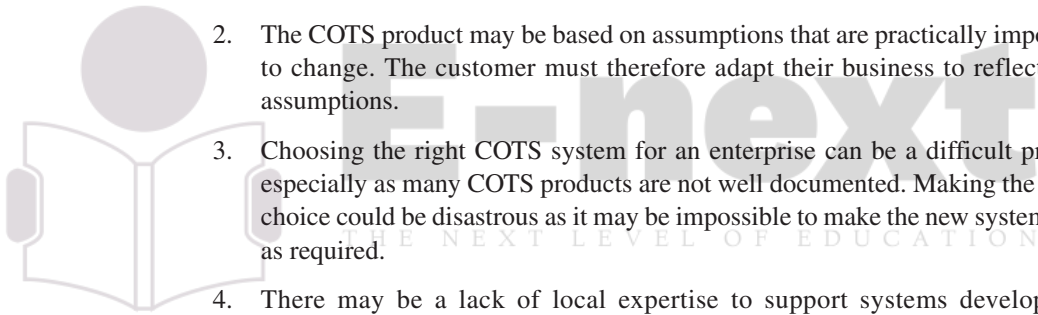
2. It is possible to see what functionality is provided by the applications and so it is easier to judge whether or not they are likely to be suitable. Other companies may already use the applications so experience of the systems is available.
3. Some development risks are avoided by using existing software. 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 COTS product 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 COTS product. This can lead to disruptive changes to existing business processes.
2. The COTS product may be based on assumptions that are practically impossible to change. The customer must therefore adapt their business to reflect these assumptions.
3. Choosing the right COTS system for an enterprise can be a difficult process, especially as many COTS products are not well documented. Making the wrong choice could be disastrous as 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 biased and geared to selling products and services, rather than meeting the real needs of the customer.
5. The COTS product vendor controls system support and evolution. They may go out of business, be taken over, or may make changes that cause difficulties for customers.

Software reuse based on COTS has become increasingly common. The vast majority of new business information processing systems are now built using COTS rather than using an object-oriented approach. Although there are often problems with this approach to system development (Tracz, 2001), success stories (Baker, 2002; Balk and Kedia, 2000; Brownsword and Morris, 2003; Pfarr and Reis, 2002) show that COTS-based reuse reduces effort and the time to deploy the system.

There are two types of COTS product reuse, namely COTS-solution systems and COTS-integrated systems. COTS-solution systems consist of a generic application from a single vendor that is configured to customer requirements. COTS-integrated systems involve integrating two or more COTS systems (perhaps from different



COTS-solution systems	COTS-integrated systems
Single product that provides the functionality required by a customer	Several heterogeneous system products are integrated to provide customized functionality
Based around 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 16.11 COTS-solution and COTS-integrated systems

vendors) to create an application system. Figure 16.11 summarizes the differences between these different approaches.

16.4.1 COTS-solution systems

COTS-solution 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 COTS-solution system may be produced for dentists that handles appointments, dental records, patient recall, etc. At a larger scale, an Enterprise Resource Planning (ERP) system may support all of the manufacturing, ordering, and customer relationship management activities in a large company.

Domain-specific COTS-solution 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 users work and these 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 in the system.

ERP systems, such as those produced by SAP and BEA, are large-scale integrated systems designed to support business practices such as ordering and invoicing, inventory management, and manufacturing scheduling (O’Leary, 2000). The configuration process for these systems involves gathering detailed information about the customer’s business and business processes, and embedding this 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

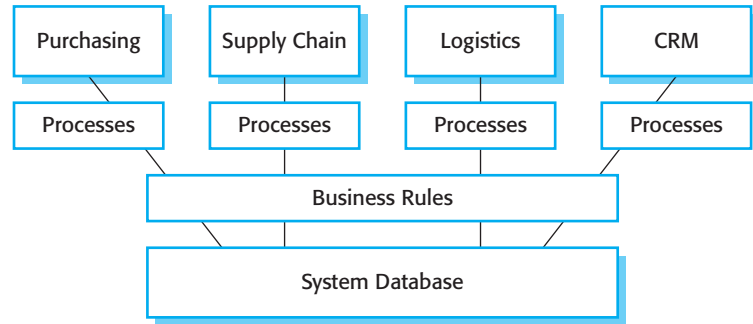


Figure 16.12 The architecture of an ERP system

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 16.12.

The key features of this architecture are:

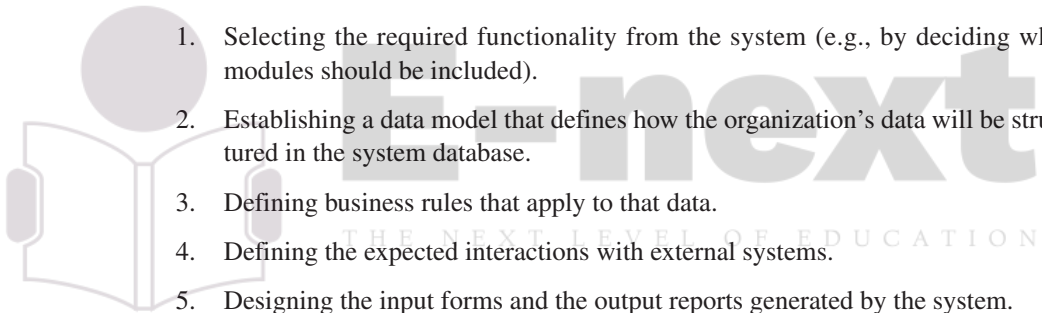
1. A number of modules to support different business functions. These are large-grain modules that may support entire departments or divisions of the business. In the example shown in Figure 16.12, the modules that have been selected for inclusion in the system are a module to support purchasing, a module 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 processes, associated with each module, which relate to activities in that module. For example, there may be a definition of the ordering process that defines 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. This means that 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, there may be a business rule 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. However, the obvious limitation of this approach to reuse is that the functionality of the system

is restricted to the functionality of the generic core. Furthermore, a company's processes and operations have to be expressed in the system configuration language, and there may be a mismatch between the concepts in the business and the concepts supported in the configuration language.

For example, in an ERP system that was sold to a university, the concept of a customer had to be defined. This caused great difficulties when configuring the system. However, universities have multiple types of customers, such as students, research funding agencies, educational charities, etc., each of which have different characteristics. None of them are really comparable to the notion of a commercial customer (i.e., a person or business that buys products or services). A serious mismatch between the business model used by the system and that of the buyer of the system makes it highly probable that the ERP system will not meet the buyer's real needs (Scott, 1999).

Both domain-specific COTS products and 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 (e.g., 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, a COTS-solution system is then ready for testing. Testing is a major problem when systems are configured rather than programmed using a conventional language. Because these systems are built using a reliable platform, obvious system failures and crashes are relatively rare. Rather the problems are often subtle and relate to the interactions between the operational processes and the system configuration. These may only be detectable by end-users and so may not be discovered during the system testing process. Furthermore, automated unit testing, supported by testing frameworks such as JUnit, cannot be used. The underlying system is unlikely to support any kind of test automation and there may be no complete system specification that can be used to derive system tests.

16.4.2 COTS-integrated systems

COTS-integrated systems are applications that include two or more COTS products or, sometimes, legacy application systems. You may use this approach when there is no single COTS system that meets all of your needs or when you wish to integrate a new COTS product with systems that you already use. The COTS products may interact through their APIs (Application Programming Interfaces) 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 COTS applications.

To develop systems using COTS products, you have to make a number of design choices:

1. *Which COTS products offer the most appropriate functionality?* Typically, there will be several COTS products available, which can be combined in different ways. If you don't already have experience with a COTS product, it can be difficult to decide which product is the most suitable.
2. *How will data be exchanged?* Different products normally use unique data structures and formats. You have to write adaptors that convert from one representation to another. These adaptors are run-time systems that operate alongside the COTS products.
3. *What features of a product will actually be used?* COTS products 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. The failure of the first flight of the Ariane 5 rocket (Nuseibeh, 1997) was a consequence of a failure in an inertial navigation system that was reused from the Ariane 4 system. However, the functionality that failed was not actually required in Ariane 5.

Consider the following scenario as an illustration of COTS 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 paperwork costs 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 e-mail system that



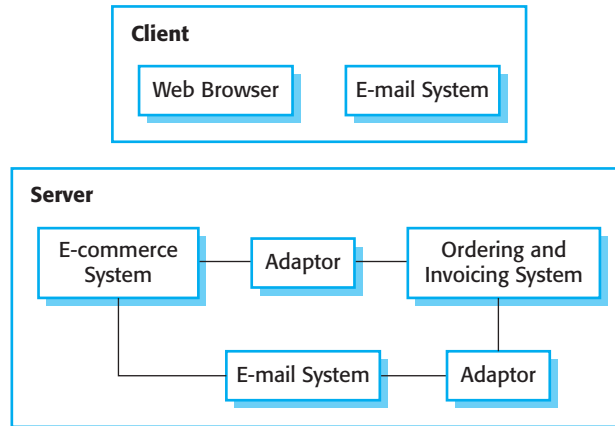


Figure 16.13 A COTS-integrated procurement system

handles communications with users. The structure of the final procurement system, constructed using COTS, is shown in Figure 16.13.

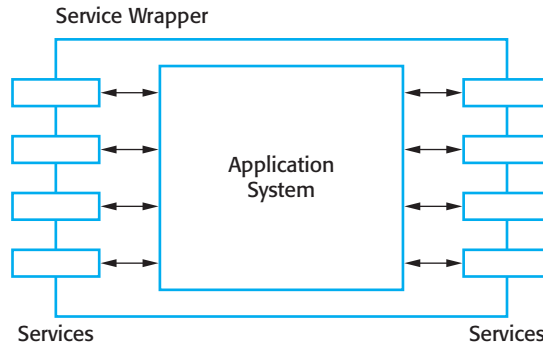
This procurement system is a client–server based and, on the client, standard web browsing and e-mail software are used. On the server, the e-commerce platform has to integrate with the existing ordering system through an adaptor. The e-commerce system has its own format for orders, confirmations of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

Months, sometimes years, of implementation effort can be saved, and the time to develop and deploy a system can be drastically reduced using a COTS-integrated approach. The procurement system described above was implemented and deployed in a very large company in nine months, rather than the three years that they estimated would be required to develop the system in Java.

COTS 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 16.14). This approach is particularly valuable for legacy systems that have to be integrated with newer application systems.

In principle, integrating COTS products is the same as integrating any other components. 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 COTS systems to operate together.

Figure 16.14
Application wrapping



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 (1999) discuss four important COTS 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, these 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 COTS product integrator but may not be of real concern for the product vendor. Users may simply have to find work-arounds to problems if they wish to reuse the COTS product.
2. *Problems with COTS system interoperability* It is sometimes difficult to get COTS products to work together because each product embeds its own assumptions about how it will be used. Garlan et al. (1995), reporting on their experience of trying to integrate four COTS products, found that three of these products were event-based but 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. (2009) concluded that the integration problems that they discovered had not been solved. Torchiano and Morisio (2004) found that lack of compliance with standards in some COTS products meant that integration was more difficult than anticipated.
3. *No control over system evolution* Vendors of COTS products 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 COTS vendors* The level of support available from COTS 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. Although vendors may commit to providing support, changing market and economic circumstances may make it difficult for them to deliver this commitment. For example, a COTS 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 all of 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 COTS-integrated systems. All of 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 real difficulties will arise with the integrated COTS products.

KEY POINTS

- Most new business software systems are now developed by reusing knowledge and code from previously implemented systems.
- 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.
- COTS product 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 COTS product or by integrating two or more COTS products.
- Enterprise Resource Planning systems are examples of large-scale COTS reuse. You create an instance of an ERP system by configuring a generic system with information about the customer's business processes and rules.
- Potential problems with COTS-based reuse include lack of control over functionality and performance, lack of control over system evolution, the need for support from external vendors, and difficulties in ensuring that systems can interoperate.

FURTHER READING

Reuse-based Software Engineering. A comprehensive discussion of different approaches to software reuse. The authors cover technical reuse issues and managing reuse processes. (H. Mili, A. Mili, S. Yacoub and E. Addy, John Wiley & Sons, 2002.)

‘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>.

‘Construction by Configuration: A New Challenge for Software Engineering’. This is an invited paper that I wrote in which 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 COTS 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>.

EXERCISES

E-next
THE NEXT LEVEL OF EDUCATION

- 16.1. What are the major technical and nontechnical factors that hinder software reuse? Do you personally reuse much software and, if not, why not?
- 16.2. Suggest why the savings in cost from reusing existing software are not simply proportional to the size of the components that are reused.
- 16.3. Give four circumstances where you might recommend against software reuse.
- 16.4. 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.
- 16.5. Using the example of the weather station system described in Chapters 1 and Chapter 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.
- 16.6. 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. If you use Microsoft Office or Open Office, these are good examples to use for this exercise.

- 16.7. 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?
- 16.8. Identify six possible risks that can arise when systems are constructed using COTS. What steps can a company take to reduce these risks?
- 16.9. Explain why adaptors are usually needed when systems are constructed by integrating COTS products. Suggest three practical problems that might arise in writing adaptor software to link two COTS application products.
- 16.10. 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

- Baker, T. (2002). 'Lessons Learned Integrating COTS into Systems'. *Proc. ICCBSS 2002 (1st Int. Conf on COTS-based Software Systems)*, Orlando, Fla.: Springer, 21–30.
- Balk, L. D. and Kedia, A. (2000). 'PPT: A COTS Integration Case Study'. *Proc. Int. Conf. on Software Eng.*, Limerick, Ireland: ACM Press, 42–9.
- Baumer, D., Gryczan, G., Knoll, R., Lilienthal, C., Riehle, D. and Zullighoven, H. (1997). 'Framework Development for Large Systems'. *Comm. ACM*, **40** (10), 52–9.
- Boehm, B. and Abts, C. (1999). 'COTS Integration: Plug and Pray?' *IEEE Computer*, **32** (1), 135–38.
- Brownsword, L. and Morris, E. (2003). 'The Good News about COTS'. <http://www.sei.cmu.edu/news-at-sei/features/2003/1q03/feature-1-1q03.htm>
- Cusamano, M. (1989). 'The Software Factory: A Historical Interpretation'. *IEEE Software*, **6** (2), 23–30.
- Fayad, M. E. and Schmidt, D. C. (1997). 'Object-oriented Application Frameworks'. *Comm. ACM*, **40** (10), 32–38.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley.
- Garlan, D., Allen, R. and Ockerbloom, J. (1995). 'Architectural Mismatch: Why Reuse is so Hard'. *IEEE Software*, **12** (6), 17–26.
- Garlan, D., Allen, R. and Ockerbloom, J. (2009). 'Architectural Mismatch: Why Reuse is Still so Hard'. *IEEE Software*, **26** (4), 66–9.
- Griss, M. L. and Wosser, M. (1995). 'Making reuse work at Hewlett-Packard'. *IEEE Software*, **12** (1), 105–7.