14 Design with reuse

Objectives

The objective of this chapter is to explain how the reuse of existing software may be incorporated into the systems design process. When you have read this chapter, you will:

- understand the benefits of reusing software components and some of the problems of reuse that can arise;
- understand different types of component that may be reused and design processes for reuse;
- have been introduced to the notion of application families and understand how application families are an effective way to reuse software;
- understand how patterns are high-level abstractions that promote design reuse in object-oriented development.

Contents

The design process in most engineering disciplines is based on component reuse. Mechanical or electrical engineers do not specify a design where every component has to be manufactured specially. They base their design on components that have been tried and tested in other systems. These components are not just small components such as flanges and valves but include major sub-systems such as engines, condensers or turbines.

It is now generally accepted that we need a comparable approach for software development. Software should be considered as an asset and reuse of these assets is essential to increase the return on their development costs. Demands for lower software production and maintenance costs, faster delivery of systems and increased quality can only be met by widespread and systematic software reuse.

To achieve software reuse, it must be considered during the software design or requirements engineering process. Opportunistic reuse is possible during programming when components are discovered that happen to fit a requirement. However, *systematic* reuse requires a design process that considers how existing designs may be reused and that explicitly organises the design around available software components.

Reuse-based software engineering is an approach to development which tries to maximise 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 either by incorporating it without change into other systems (COTS product reuse, section 14.1.2) or by developing application families that may run on different platforms or may be specialised to the needs of particular customers (application families, section 14.2).

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. This is covered in section 14.3.

3. *Function reuse* Software components which implement a single function, such as a mathematical function, may be reused. This form of reuse, based around standard libraries, has been common for the past 40 years.

Application system reuse has been widely practised for many years as software companies implement their systems across a range of machines and tailor them for different environments. Function reuse is also well established through standard libraries of reusable functions such as graphics and mathematical libraries. However, although there has been interest in component reuse since the early 1980s, it is only in the past few years that it has become accepted as a practical approach to software systems development.

An obvious advantage of software reuse is that overall development costs should be reduced. Fewer software components need be specified, designed, implemented and validated. However, cost reduction is only one potential advantage of reuse. There are a number of other advantages to reusing software assets, as shown in Figure 14.1.

Benefit	Explanation
Increased reliability	Reused components that have been exercised in working systems should be more reliable than new components. They have been tried and tested in a variety of different environments. Design and implementation faults are discovered and eliminated in the initial use of the components, thus reducing the number of failures when the component is reused.
Reduced process risk	If a component exists, there is less uncertainty in the costs of reusing that component than in the costs of development. This is an important factor for project management as it reduces the uncertainties in project cost estimation. This is particularly true when relatively large components such as sub-systems are reused.
Effective use of specialists	Instead of application specialists doing the same work on different projects, these specialists can develop reusable components which encapsulate their knowledge.
Standards compliance	Some standards, such as user interface standards, can be implemented as a set of standard components. For example, reusable components may be developed to implement menus in a user interface. All applications present the same menu formats to users. The use of standard user interfaces improves reliability as users are less likely to make 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 components speeds up system production because both development and validation time should be reduced.

Figure 14.1 Benefits of software reuse

There are three critical requirements for software design and development with reuse:

1. It must be possible to find appropriate reusable components. Organisations need a base of properly catalogued and documented reusable components. It must be easy to find components in this catalogue if they exist.
2. The reuser of the components must be confident that the components will behave as specified and will be reliable. Ideally, all components in an organisation's catalogue should be certified to confirm that they have reached some quality standards. In practice, this is unrealistic and people in a company learn informally about reliable components.
3. The components must have associated documentation to help the reuser understand them and adapt them to a new application. The documentation should include information about where components have been reused and any reuse problems which have been found.

The successful use of Visual Basic and Visual C++ with components and Java with JavaBeans has demonstrated the value of reuse. Component-based software

Problem	Explanation
Increased maintenance costs	If component source code is not available then maintenance costs may be increased, as the reused elements of the system may become increasingly incompatible with system changes.
Lack of tool support	CASE toolsets 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.
Not-invented-here syndrome	Some software engineers sometimes prefer to rewrite components as they believe that they can improve on the reusable component. 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.
Maintaining a component library	Populating a component library and ensuring that software developers can use this library can be expensive. Our current techniques for classifying, cataloguing and retrieving software components are immature.
Finding 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 will routinely include a component search as part of their normal development process.

Figure 14.2 Problems with reuse

engineering (Szyperski, 1998) is becoming widely accepted as a cost-effective approach to software development.

However, there are some costs and problems associated with reuse (Figure 14.2). These may inhibit the introduction of reuse and may mean that the reductions in overall development cost through reuse may be less than anticipated.

These difficulties mean that systematic reuse does not just happen but must be planned and introduced through an organisation-wide reuse programme. This has been recognised for many years in Japan (Matsumoto, 1984) where reuse is an integral part of the Japanese 'factory' approach to software development (Cusamano, 1989). Companies such as Hewlett-Packard have also been very successful in their reuse programmes (Griss and Wosser, 1995) and their experience has been incorporated in a general book by Jacobson *et al.* (1997).

An alternative to the component-oriented view of reuse is the generator view. In this approach to reuse, reusable knowledge is captured in a program generator system which can be programmed in a domain-oriented language. The application description specifies, in an abstract way, which reusable components are to be used, how they are to be combined and their parameterisation. Using this information, an operational software system can be generated (Figure 14.3).

Generator-based reuse is only possible when domain abstractions and their mapping to executable code can be identified. A domain-specific language (such as a 4GL) is then used to compose and control the domain abstractions. Areas where this has been successful include:

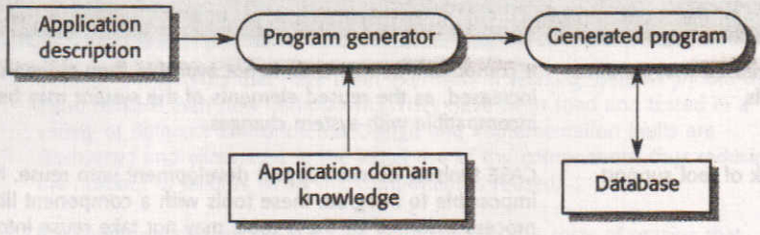


Figure 14.3 Generator-based reuse

1. *Application generators for business data processing* The input to these may be a 4GL or may be completely interactive where the user defines screens and processing actions. The output is a program in a language such as COBOL or SQL.
2. *Parser generators for language processing* The generator input is a grammar describing the language to be parsed and the output is a language parser.
3. *Code generators in CASE tools* The input to these generators is a software design and the output is a program implementing the designed system.

Generator-based reuse is cost-effective but depends on identifying stereotypical domain abstractions. This has been possible in the above areas and, to a lesser extent, in areas such as command and control systems (O'Connor *et al.*, 1994). Its prime advantage is that it is easier for end-users to develop programs using generators compared to other component-based approaches to reuse. However, the need for a deep understanding of application domain concepts and domain models has limited the applicability of this technique.

14.1 Component-based development

Component-based development or component-based software engineering emerged in the late 1990s as a reuse-based approach to software systems development. Its motivation was frustration that object-oriented development had not led to extensive reuse as originally suggested. Single object classes were too detailed and specific and had to be bound with an application either at compile-time or when the system was linked. Detailed knowledge of the classes were required to use them and this usually meant that source code had to be available. This presented difficult problems for marketing components. In spite of early optimistic predictions, no significant market for individual objects has ever developed.

Components are more abstract than object classes and can be considered to be stand-alone service providers. When a system needs some service, it calls on

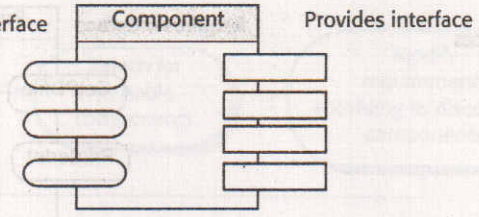


Figure 14.4 Component interfaces

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 very simple component could be a single mathematical function that computes the square root of a number. When a program requires a square root computation, it calls on that component to provide it. At the other end of the scale, a system that needs to carry out some arithmetic computations could call on a spreadsheet component that provides a calculation service.

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

1. The component is an independent executable entity. Source code is not available so that the component is not compiled with other system components.
2. Components publish their interface and all interactions are through that interface. The component interface is expressed in terms of parameterised operations and its internal state is never exposed.

Components are defined by their interfaces and, in the most general cases, can be thought of as having two related interfaces as shown in Figure 14.4:

- A *provides* interface that defines the services provided by the component
- A *requires* interface that specifies what services must be available from the system that is using the component. If these are not provided then the component will not work.

For example, consider a component that provides printing services as illustrated in Figure 14.5. In this case, the services that are provided are services to print a document, to discover the state of the queue associated with a particular printer, to register and unregister a printer with the printing services component, to move a print job from one printer to another and to remove a job from a print queue. The requirements for this component are that the underlying platform should provide a service called GetPDFile to retrieve the printer description file for a printer type and a service called PrinterInt that transfers commands to a specified printer.

Components may exist at different levels of abstraction, from a simple library subroutine to an entire application such as Microsoft Excel. Meyer (1999) identifies five distinct levels of abstraction:

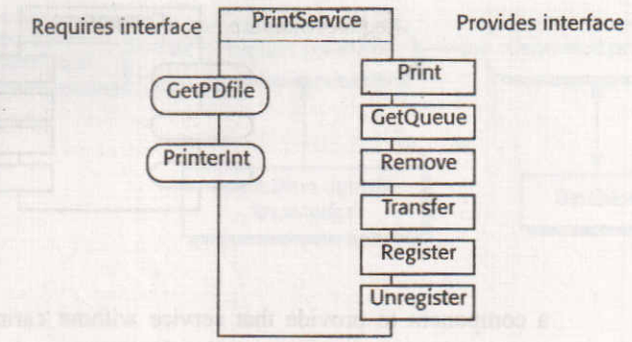


Figure 14.5
A printing services component

1. *Functional abstraction* The component implements a single function such as a mathematical function. In essence, the *provides* interface is the function itself.
2. *Casual groupings* The component is a collection of loosely related entities that might be data declarations, functions, etc. The *provides* interface consists of the names of all of the entities in the grouping.
3. *Data abstractions* The component represents a data abstraction or class in an object-oriented language. The *provides* interface consists of operations to create, modify and access the data abstraction.
4. *Cluster abstractions* The component is a group of related classes that work together. These are sometimes called frameworks. The *provides* interface is the composition of all of the provides interfaces of the objects that make up the framework. I discuss frameworks in section 14.1.1.
5. *System abstraction* The component is an entire self-contained system. Reusing system-level abstractions is sometimes called COTS product reuse. The *provides* interface is the so-called API (Application Programming Interface) that is defined to allow programs to access the system commands and operations. I discuss COTS product reuse in section 14.1.2.

Component-oriented development can be integrated into a systems development process by incorporating a specific reuse activity as shown in Figure 14.6. The system designer completes a high-level design and specifications of the components of that design. These specifications are used to find components to reuse. These may be incorporated at the architectural level or at more detailed design levels.

Although this approach can result in significant reuse, it contrasts with the approach adopted in other engineering disciplines where reusability drives the design process. Rather than design then search for reusable components, engineers first search for

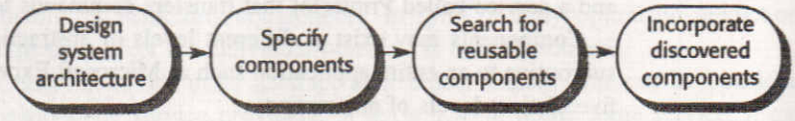


Figure 14.6
An opportunistic reuse process

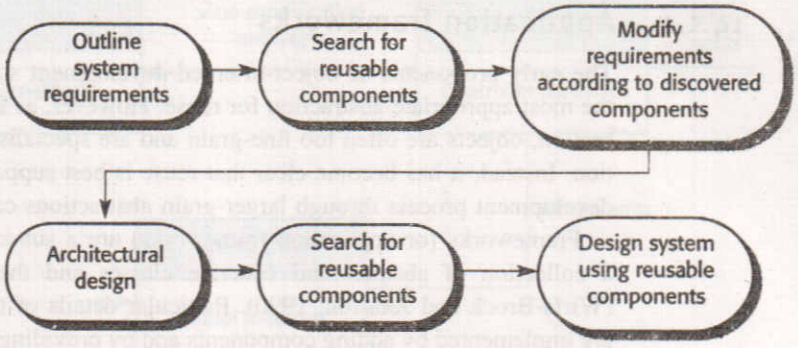


Figure 14.7 Development with reuse

reusable components. They base their design on the components that are available (Figure 14.7).

In reuse-driven development, the system requirements are modified according to the reusable components available. The design is also based around existing components. Of course, this means that there may have to be requirements compromises. The design may be less efficient than a special-purpose design. However, the lower costs of development, more rapid system delivery and increased system reliability should compensate for this.

The process of implementing the system using components is usually either a prototyping process or an incremental development process. A standard programming language such as Java may be used with components in a library referenced from the program. Alternatively (and more commonly) a scripting language that is specifically designed for integrating reusable components is used to support rapid program development.

The first scripting language that was proposed for reusable component integration was the Unix shell (Bourne, 1978). Since then, a number of scripting languages, such as Visual Basic and TCL/TK, have been developed. Ousterhout (1998) discusses the benefits of scripting languages, including their typeless nature and the fact that they are interpreted rather than compiled.

As I discussed in Chapter 8 (Software Prototyping), users of Visual Basic construct applications visually by defining the interface by selecting types of interface component such as menus, text input fields, check boxes, etc. and positioning them on the screen. Individual components or component sequences that carry out some computation may then be associated with the interface entities.

Perhaps the main difficulty with component-based development is the problem of maintenance and evolution. Typically, the source code of components is not available and, as application requirements change, it may be impossible to change the components to reflect these requirements. The option of changing requirements at this stage to fit available components is not usually possible as the application is already in use. Additional work is therefore required to reuse components and, over time, this leads to increased maintenance costs. However, given that component-based development allows for faster delivery of software, organisations may be willing to accept these longer-term costs.

14.1.1 Application frameworks

The early proponents of object-oriented development suggested that objects were the most appropriate abstraction for reuse. However, as I explained in the previous section, objects are often too fine-grain and are specialised to a particular application. Instead, it has become clear that reuse is best supported in an object-oriented development process through larger-grain abstractions called frameworks.

Frameworks (or application frameworks) are a sub-system design made up of a collection of abstract and concrete classes and the interface between them (Wirfs-Brock and Johnson, 1990). Particular details of the application sub-system are implemented by adding components and by providing concrete implementations of abstract classes in the framework. Frameworks are rarely applications in their own right. Applications are normally constructed by integrating a number of frameworks.

Fayad and Schmidt (1997) have identified three classes of framework:

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 CORBA, Microsoft's COM and DCOM and Java Beans (Orfali and Harkey, 1998). I have already described this type of framework in Chapter 11 where I covered distributed object architectures.
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. These frameworks are related to application families whose structure I discuss in section 14.2. However, they are usually more abstract and thus allow a wider range of applications to be created.

As the name suggests, a framework is a generic structure that can be extended to create a more specific sub-system or application. Extending the framework may involve adding concrete classes that inherit operations from abstract classes in the framework. In addition, call-backs may be defined. These are methods that are called in response to events that are recognised by the framework.

At the time of writing, the best developed frameworks are system infrastructure frameworks, especially those concerned with graphical user interfaces. Enterprise application frameworks are starting to emerge (Codenie *et al.*, 1997) but we are still trying to understand the most effective structures and organisations for these frameworks.

One of the best known and widely used frameworks for GUI design is the Model-View-Controller framework (Figure 14.8). This was originally proposed in the 1980s as an approach to GUI design that allowed for multiple presentations of

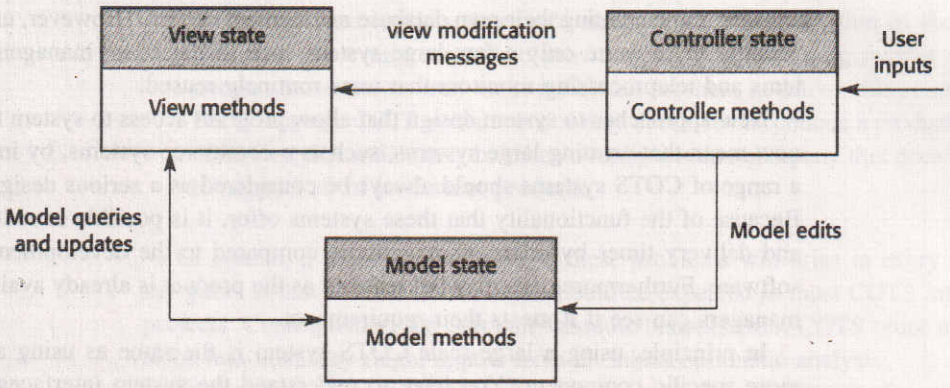


Figure 14.8 The Model-View-Controller framework

an object and separate styles of interaction with each of these presentations. The MVC framework supports the presentation of data in different ways (see Figure 14.13) and separate interaction with each of these presentations. When the data is modified through one of the presentations, all of the other presentations are updated.

Frameworks are often instantiations of a number of patterns as discussed in section 14.2. For example, the MVC framework includes the Observer pattern which is described in Figure 14.12, the Strategy pattern which is concerned with updating the model, the Composite pattern and a number of others that are discussed by Gamma *et al.* (1995).

Applications that are constructed using frameworks can be the basis for further reuse through the concept of application families as discussed in section 14.3. Because these applications are constructed using a framework, modifying family members to create new family members is simplified.

The fundamental problem with frameworks is their inherent complexity and the time it takes to learn to use them. Several months may be required to completely understand a framework so it is likely that, in large organisations, some software engineers will become framework specialists. There is no doubt that this is an effective approach to reuse but the high cost of introduction limits its widespread use.

14.1.2 COTS product reuse

The term COTS (Commercial-Off-The-Shelf) products can, in principle, apply to any component that is offered by a third-party vendor. However, it is more normally used to refer to system software products. Hence we refer to COTS systems or sometimes just COTS (Commercial Off-the-shelf Systems). I normally prefer to talk about COTS systems. As the functionality offered by these systems is much wider than that of more specialised components, the potential payoff through reuse is increased.

Of course, some types of COTS system have been reused for many years. Database systems are perhaps the best example of this. Very few developers would

consider implementing their own database management system. However, until fairly recently, there were only a few large systems such as database management systems and teleprocessing monitors that were routinely reused.

New approaches to system design that allow program access to system functions now mean that creating large systems, such as e-commerce systems, by integrating a range of COTS systems should always be considered as a serious design option. Because of the functionality that these systems offer, it is possible to reduce costs and delivery times by orders of magnitude compared to the development of new software. Furthermore, risks may be reduced as the product is already available and managers can see if it meets their requirements.

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

However, the fact that these products are usually large systems in their own right and are often sold as separate stand-alone systems introduces additional problems. Boehm (1999) discusses four problems with COTS system integration:

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 operation. Fixing these problems may be a priority for the COTS product integrator but may not be of real concern to 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 different assumptions about how it will be used. Garlan *et al.* (1995) reported on their experience of trying to integrate four COTS products and found that three of these products were event-based but each used a different model of events and assumed that it had exclusive access to the event queue. As a consequence, the project required five times as much effort as originally predicted and the schedule was two years rather than the predicted six months.

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 very 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 that is available from COTS vendors varies widely. Because these are off-the-shelf systems, vendor support is particularly important when problems arise because 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 COTS system vendor may decide to discontinue a product because of limited demand or may be taken over by another company that does not wish to support all of its current products.

Of course, it is unlikely that all of these problems will arise in every case but my guess is that at least one of them should be expected in most COTS integration projects. Consequently, the cost and schedule benefits from COTS reuse are likely to be less than they might appear from an initial optimistic analysis.

Furthermore, Boehm (1999) reckons that, in many cases, the cost of system maintenance and evolution may be greater when COTS products are used. All of the above difficulties are life-cycle problems and they don't just affect the initial development of the system. As the people involved in the system maintenance become more remote from the original system developers, the more likely it is that real difficulties will arise with the integrated COTS products.

In spite of these problems, the benefits of COTS product reuse are very significant because these systems offer so much functionality to the reuser. Months and sometimes years of implementation effort can be saved if an existing system is reused and system development times drastically reduced. Given that rapid system delivery is the key requirement for many systems, this form of reuse is likely to become more and more widely practised.

14.1.3 Component development for reuse

The ideal component development process should be an experience-based process where reusable components are specially constructed from existing components that have already been reused in an opportunistic way. By using knowledge of reuse problems and component adaptations required to support reuse, a more generic and hence more reusable version of the component can be created.

There are various component characteristics that lead to reusability:

1. The component should reflect stable domain abstractions. Stable domain abstractions are fundamental concepts in the application domain that change slowly. For example, in a banking system, domain abstractions might be accounts, account holders, statements, etc. In a hospital management system, domain abstractions might be patients, treatments, nurses, etc.
2. The component should hide the way in which its state is represented and should provide operations that allow the state to be accessed and updated. For example, in a component that represents a bank account, there should be operations to query the balance of the account, change the account balance, record transactions on the account, etc.

3. The component should be as independent as possible. Ideally, a component should be stand-alone so that it does not need any other components to operate. In practice, this is only possible for very simple components, and more complex components will inevitably have some dependencies on other components. It is best to minimise these, especially if they are dependencies on components, such as operating system functions, that may change.

4. All exceptions should be part of the component interface. Components should not handle exceptions themselves as different applications will have different requirements for exception handling. Rather, the component should define what exceptions can arise and should publish these as part of the interface. For example, a simple component implementing a stack data structure should detect and publish stack overflow and stack underflow exceptions.

In many existing systems there are large segments of code that implement domain abstractions but cannot be used directly as components. The reason for this is that they are not packaged according to the model in Figure 14.4 with clearly defined *requires* and *provides* interfaces. To make these components reusable, it is usually necessary to construct a wrapper. The wrapper hides the complexity of the underlying code and provides an interface for external components to access services that are provided.

There is an inevitable trade-off between reusability and usability of a component. To make the component reusable implies providing a very general interface with operations that cater for different ways in which the component may be used. To make the component usable implies providing a simple, minimal interface that is easy to understand. Reusability adds complexity and hence reduces component understandability. It is therefore more difficult for engineers to decide when and how to reuse that component. Designers of reusable components must therefore find a compromise between generality and understandability.

14.2 Application families

One of the most effective approaches to reuse is based around the notion of application families. An application family or product line is a related set of applications that has a common domain-specific architecture as discussed in Chapter 10. However, each specific application is specialised in some way. The common core of the application family is reused each time that a new application is required. The new development may involve writing some additional components and adapting some of the components in the application to meet new demands.

There are various types of specialisation of an application family that may be developed:

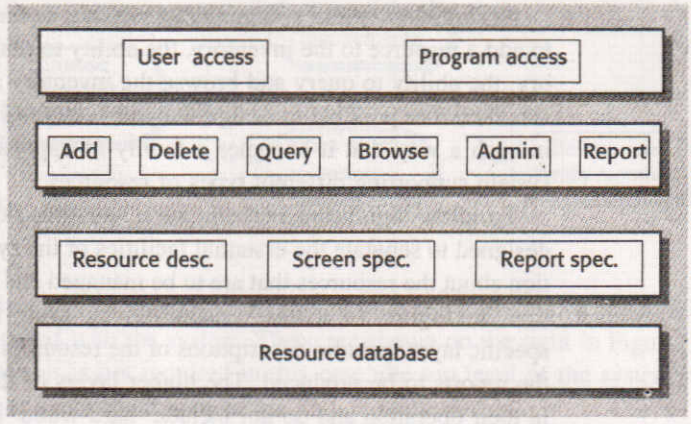


Figure 14.9
A generic resource management system

1. *Platform specialisation* where different versions of the application are developed for different platforms. For example, versions of the application may exist for Windows NT, Solaris 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. *Configuration specialisation* where different versions of the application are created to handle different peripheral devices. For example, a system for the emergency services may exist in different versions depending on the type of radio system used. In this case, the functionality may vary to reflect the functionality of the peripherals, and components that interface with peripherals must be modified.
3. *Functional specialisation* where different versions of the application are created for customers with 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.

To illustrate this approach to reuse, consider Figure 14.9 which illustrates the architecture of a system for inventory management. Inventory management systems are used by organisations to keep track of their assets. Therefore, a power utility might have an inventory management system that tracks all fixed installations and the equipment installed at these installations. A university might have an inventory management system to keep track of the equipment used in its teaching labs.

Inventory management systems obviously vary depending on the type of resources that are being managed and the information that is required for each resource. For example, the power utility system will not need to have a facility that allows resource locations to be changed – the nature of this system is that these are fixed. The university inventory system, however, must be able to change the location of equipment as it is moved from one lab to another.

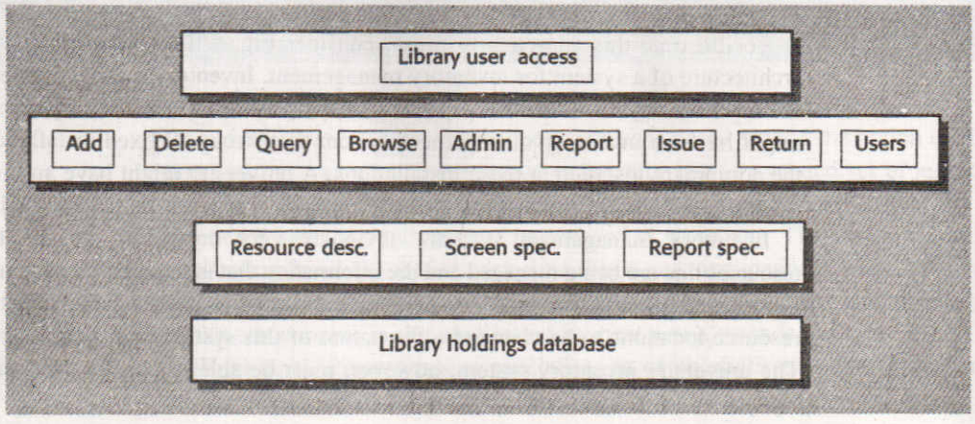
Nevertheless, these systems must provide a core of facilities such as the ability to add a resource to the inventory, the ability to remove a resource from the inventory, the ability to query and browse the inventory and the ability to create reports. It is therefore possible to design the architecture of inventory management systems in such a way that it becomes a family of applications with each instance of the system supporting different types of resources.

To allow significant reuse in such situations, the system architecture must be designed to separate the essential facilities of the system from the detailed information about the resources that are to be managed and the user access to such information. In Figure 14.9, this is achieved by using a layered architecture where a specific layer includes descriptions of the resources, the screens to be displayed and the reports to be produced. The higher layers of the system use these descriptions in their operation and do not include 'hard-wired' information about the resources. Different inventory management applications may be produced by modifying this description layer.

Of course, this type of tailoring may be achieved in object-oriented systems by specifying an abstract resource object and then using the inheritance facilities to specialise this depending on the type of resource that is being managed. This would probably lead to a rather different architecture from that shown in Figure 14.9. However, for this type of system, object-oriented development may not be appropriate. When applications rely on large databases with millions of records but relatively few types of logical entity, an object-oriented system is likely to be significantly less efficient than a system built around a relational database. At the time of writing, commercial object-oriented databases are still relatively slow and are not suited to supporting hundreds of transactions per second.

As well as developing new members of the application family by developing new resource descriptions, it is also possible to add new functionality to the system by including new modules in the system layers. To illustrate this, I have adapted the resource management system in Figure 14.9 to create a library system (Figure 14.10).

Figure 14.10
A library system



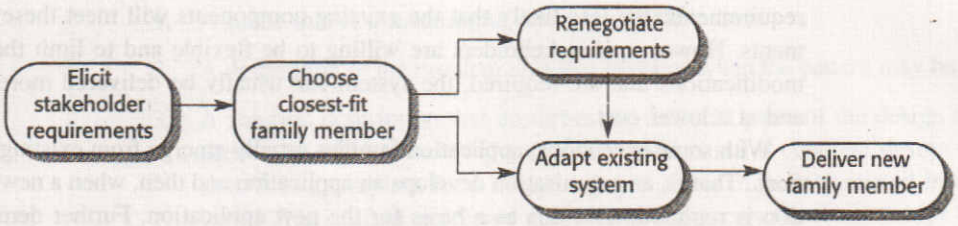


Figure 14.11 Family member development

This involves adding new facilities to issue and return resources and to allow users to be registered with the system. These are shown on the right in Figure 14.10. As program access is not required in this case, the top level of the system need only support user access to the resources.

In general, developing applications by adapting a generic version of the application means that a very high proportion of the application code is reused. Furthermore, application experience is often transferable from one system to another so that 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 and the overall development time for the application is reduced.

Figure 14.11 shows the steps involved in adapting an application family to create a new application. The details of the process are likely to differ radically depending on the application domain and the system's organisational environment. The steps involved in the generic process are:

1. *Elicit stakeholder requirements* This may be based on a normal requirements engineering process. However, as a system already exists, it will normally involve demonstrating and experimenting with that system and expressing the requirements as modifications that are required.
2. *Choose closest-fit family member* The requirements are analysed and the family member that is the closest fit to these is chosen for modification. This need not be the system that was demonstrated.
3. *Renegotiate requirements* As more details of the changes required to the existing system emerge and the project is planned, there may be some requirements renegotiation to minimise 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 member of the application family 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 an application family 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 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, the system can usually be delivered more quickly and at a lower cost.

With some exceptions, application families usually emerge from existing applications. That is, an organisation develops an application and then, when a new application is required, uses this as a basis for the new application. Further demands for new applications cause the process to continue. However, as change tends to corrupt application structure, at some stage a specific decision to design a generic application family is made. This design is based on reusing the knowledge that has been gained from developing the initial set of applications.

14.3 Design patterns

When you try to reuse executable components you are inevitably constrained by detailed design decisions that have been made by the implementors of these components. These range from the particular algorithms that have been used to implement the components to the objects and types in the component interfaces. If these design decisions conflict with your particular requirements then reusing the component is either impossible or introduces significant inefficiencies into your system.

One way round this is to reuse more abstract designs that do not include implementation detail. These are then implemented specifically to fit your application requirements. The first instances of this approach to reuse came in the documentation and publication of fundamental algorithms (Knuth, 1971) and, later, in the documentation of abstract data types such as stacks, trees and lists (Booch, 1987). More recently, this approach to reuse has been embodied in the notion of design patterns.

Design patterns (Gamma *et al.*, 1995) were derived from ideas put forward by Christopher Alexander (Alexander *et al.*, 1977), who suggested that there were certain patterns of building design that were common and that were inherently pleasing and effective. The 'pattern' is a description of the problem and the essence of its solution so that the solution may be reused in different settings. The pattern is not a detailed specification. Rather, you can think of it as a description of accumulated wisdom and experience. It is a well-tried solution to a common problem. In this respect, patterns may be used during analysis to develop system models as well as during the design process.

In software design, design patterns have been inevitably associated with object-oriented design. They often rely on object characteristics such as inheritance and polymorphism to provide generality. However, the general principle is one that is equally applicable to all approaches to software design.

Gamma *et al.* (1995) define the four essential elements of a design patterns:

1. A name that is a meaningful reference to the pattern.
2. A description of the problem area that explains when the pattern may be applied.
3. A solution description that describes the different parts of the design solution, their relationships and responsibilities. This is not a concrete design description. It is a template for a design solution that can be instantiated in different ways. This is often expressed graphically and shows the relationships between the objects and object classes in the solution.
4. A statement of the consequences – the results and trade-offs – of applying the pattern. This is used to help designers understand whether or not a pattern can be effectively applied in a particular situation.

These essential elements of a pattern description may themselves be further decomposed. For example Gamma *et al.* (1995) break down the problem description into *motivation* – a description of why the pattern is useful, and *applicability* – a description of situations where the pattern may be used. Under the description of the solution, they describe the pattern structure, participants, collaborations and implementation.

To illustrate pattern description I have described one of the most commonly used patterns suggested by Gamma *et al.*, namely the Observer pattern (Figure 14.12). This pattern is used when different presentations of an object's state are required.

Figure 14.12
Description of the Observer pattern

Pattern name: Observer

Description: Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.

Problem description: In many situations, it is necessary to provide multiple displays of some state information such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations may support interaction and, when the state is changed, all displays must be updated.

This pattern may be used in all situations where more than one display format for state information may be required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.

Solution description: The structure of the pattern is shown in Figure 14.14. This defines two abstract objects – Subject and Observer, and two concrete objects – ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The state to be displayed is maintained in ConcreteSubject which also inherits operations from Subject allowing it to add and remove Observers and to issue a notification when the state has changed.

The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update () interface of Observer which allows these copies to be kept in step. The ConcreteObserver automatically displays its state – this is not normally an interface operation.

Consequences: The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimisations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

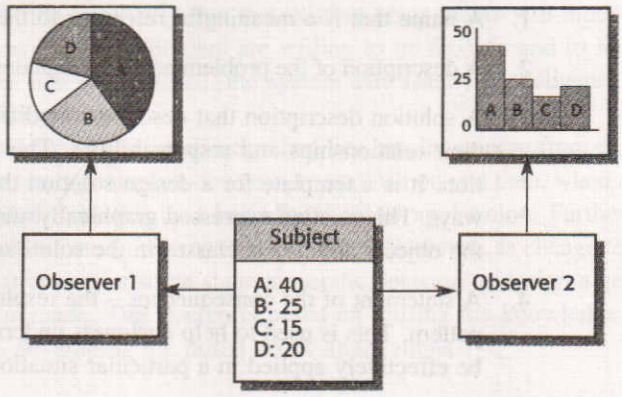


Figure 14.13
Multiple displays

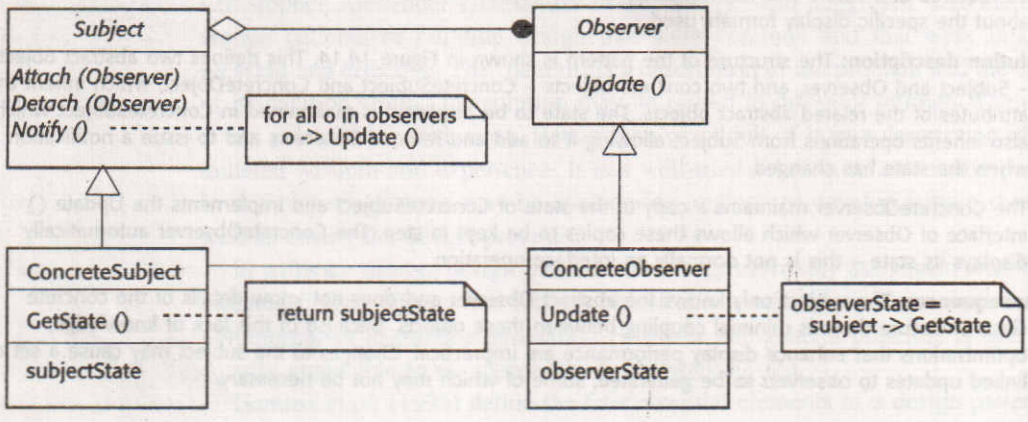
It separates the object that must be displayed and the different forms of presentation. This is illustrated in Figure 14.13 which shows two graphical presentations of the same data set. In my description, I use the four essential description elements and supplement these with a brief statement of what the pattern can do.

Graphical representations are normally used to illustrate the object classes that are used in patterns and their relationships. Figure 14.14 is the representation in UML of the Observer pattern.

The use of patterns is a very effective form of reuse but, in my opinion, they have a high cost of introduction into design processes and they can only be used effectively by experienced programmers. The reason for the high cost of introduction is that patterns are complex. To use them effectively, a designer has to know and understand many patterns in detail. This is unlike an executable component where only the interfaces need be understood. Obviously, developing this understanding takes some time.

Patterns are only suitable for use by experienced programmers because they can recognise generic situations where a pattern can be applied. Inexperienced programmers, even if they have read the pattern books, will always find it hard to decide if a pattern should be reused or if they need a special-purpose solution.

Figure 14.14 The Observer pattern



KEY POINTS

- Design with reuse involves designing the software around existing examples of good design and making use of software components where these are available.
- The advantages of software reuse are lower costs, faster software development and lower risks. System reliability is increased and specialists can be used more effectively by concentrating their expertise on the design of reusable components.
- Component-based development relies on using 'black-box' components with clearly defined 'requires' and 'provides' interfaces. Types of component that may be reused include functions, data abstractions, frameworks and complete application systems.
- 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.
- Software components that have been designed for reuse should be independent, should reflect stable domain abstractions, should provide access to state through interface operations and should not handle exceptions themselves.
- Application families are related applications that are developed from one or more base applications. A generic system is adapted and specialised to meet specific requirements for functionality, target platform or operational configuration.
- Design patterns are high-level abstractions that document successful design solutions. They are fundamental to design reuse in object-oriented development. A pattern description should include a pattern name, a problem and solution description, and a statement of the results and trade-offs of using the pattern.

FURTHER READING

Component Software: Beyond Object-oriented Programming. At the time of writing, this is the only available book on component-based development, although I expect that others will appear in the near future. It is quite a good description of the approach but is not really accessible unless you have some background in object-oriented development. (C. Szyperski, 1999, Addison Wesley Longman.)

Software Reuse: Architecture, Process and Organisation for Business Success. A comprehensive discussion of reuse with a particular emphasis on organisational issues and integrating reuse with an object-oriented development process. (I. Jacobson, M. Griss and P. Jonsson, 1997, Addison-Wesley.)

Design Patterns: Elements of Reusable Object-oriented Software. This is the original software patterns handbook that introduced the notion of software patterns to a wide community. (E. Gamma, R. Helm, R. Johnson and J. Vlissides, 1995, Addison-Wesley.)

EXERCISES

14.1 What are the major technical and non-technical factors which hinder software reuse? From your own experience, do you reuse much software? If not, why not?

14.2 Explain why the savings in cost from reusing existing software is not simply proportional to the size of the components that are reused.

14.3 Give four circumstances where you might recommend against software reuse.

14.4 Suggest possible requires and provides interfaces for the following components:

- A component implementing a bank account.
- A component implementing a language-independent keyboard. Keyboards in different countries have different key organisations and different character sets.
- A component that implements version management facilities as discussed in Chapter 29.

14.5 What is the difference between an application framework and a COTS product as far as reuse is concerned? Why is it sometimes easier to reuse a COTS product than an application framework?

14.6 Using the example of the weather station system described in Chapter 12, suggest an architecture for a family of applications that are concerned with remote monitoring and data collection.

14.7 Using the example of an inventory management family shown in Figure 14.9, suggest operations that have to be added or changed to support the reordering of inventory items when the level falls below some specified number.

14.8 Why are patterns an effective form of design reuse? What are the disadvantages to this approach to reuse?

14.9 The reuse of software raises a number of copyright and intellectual property issues. If a customer pays a software contractor to develop some 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.