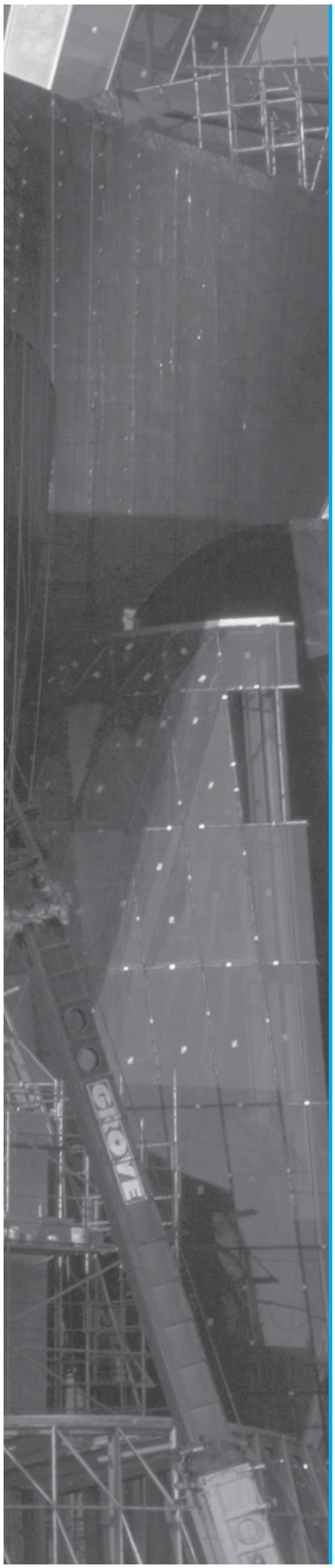

21

Aspect-oriented software engineering

Objectives

The objective of this chapter is to introduce you to aspect-oriented software development, which is based on the separation of concerns. When you have read this chapter, you will:

- understand why the separation of concerns is a good guiding principle for software development;
- have been introduced to the fundamental ideas underlying aspects and aspect-oriented software development;
- understand how an aspect-oriented approach may be used for requirements engineering, software design, and programming;
- be aware of the difficulties of testing aspect-oriented systems.

Contents

In most large systems, the relationships between the requirements and the program components are complex. A single requirement may be implemented by a number of components and each component may include elements of several requirements. In practice, this means that implementing a change to the requirements may involve understanding and changing several components. Alternatively, a component may provide some core functionality but also include code that implements several system requirements. Even when there appears to be significant reuse potential, it may be expensive to reuse such components. Reuse may involve modifying them to remove extra code that is not associated with the core functionality of the component.

Aspect-oriented software engineering (AOSE) is an approach to software development that is intended to address this problem and so make programs easier to maintain and reuse. AOSE is based around abstractions called aspects, which implement system functionality that may be required at several different places in a program. Aspects encapsulate functionality that cross-cuts and coexists with other functionality that is included in a system. They are used alongside other abstractions such as objects and methods. An executable aspect-oriented program is created by automatically combining (weaving) objects, methods, and aspects, according to specifications that are included in the program source code.

An important characteristic of aspects is that they include a definition of where they should be included in a program, as well as the code implementing the cross-cutting concern. You can specify that the cross-cutting code should be included before or after a specific method call or when an attribute is accessed. Essentially, the aspect is woven into the core program to create a new augmented system.

The key benefit of an aspect-oriented approach is that it supports the separation of concerns. As I explain in Section 21.1, separating concerns into independent elements rather than including different concerns in the same logical abstraction is good software engineering practice. By representing cross-cutting concerns as aspects, these concerns can be understood, reused, and modified independently, without regard for where the code is used. For example, user authentication may be represented as an aspect that requests a login name and password. This can be automatically woven into the program wherever authentication is required.

Say you have a requirement that user authentication is required before any change to personal details is made in a database. You can describe this in an aspect by stating that the authentication code should be included before each call to methods that update personal details. Subsequently, you may extend the requirement for authentication to all database updates. This can easily be implemented by modifying the aspect. You simply change the definition of where the authentication code is to be woven into the system. You do not have to search through the system looking for all occurrences of these methods. You are therefore less likely to make mistakes and introduce accidental security vulnerabilities into your program.

Research and development in aspect-orientation has primarily focused on aspect-oriented programming. Aspect-oriented programming languages such as AspectJ (Colyer and Clement, 2005; Colyer et al., 2005; Kiczales, et al., 2001; Laddad, 2003a; Laddad, 2003b) have been developed that extend object-oriented programming to include aspects. Major companies have used aspect-oriented programming

in their software production processes (Colyer and Clement, 2005). However, cross-cutting concerns are equally problematic at other stages of the software development process. Researchers are now investigating how to utilize aspect-orientation in system requirements engineering and system design, and how to test and verify aspect-oriented programs.

I have included a discussion of AOSE here because its focus on separating concerns is an important way of thinking about and structuring a software system. Although some large-scale systems have been implemented using an aspect-oriented approach, the use of aspects is still not part of mainstream software engineering. As with all new technologies, advocates focus on the benefits rather than the problems and costs. Although it will be some time before AOSE is routinely used alongside other approaches to software engineering, the idea of separating concerns that underlies AOSE are important. Thinking about the separation of concerns is a good general approach to software engineering.

In the remaining sections of the chapter, I therefore focus on the concepts that are part of AOSE and discuss the advantages and disadvantages of using an aspect-oriented approach at different stages of the software development process. As my aim is to help you understand the concepts underlying AOSE, I do not go into detail of any specific approach or aspect-oriented programming language.

21.1 The separation of concerns

The separation of concerns is a key principle of software design and implementation. It means that you should organize your software so that each element in the program (class, method, procedure, etc.) does one thing and one thing only. You can then focus on that element without regard for the other elements in the program. You can understand each part of the program by knowing its concern, without the need to understand other elements. When changes are required, they are localized to a small number of elements.

The importance of separating concerns was recognized at an early stage in the history of computer science. Subroutines, which encapsulate a unit of functionality, were invented in the early 1950s and subsequent program structuring mechanisms such as procedures and object classes have been designed to provide better mechanisms for realizing the separation of concerns. However, all of these mechanisms have problems in dealing with certain types of concern that cut across other concerns. These cross-cutting concerns cannot be localized using structuring mechanisms such as objects or functions. Aspects have been invented to help manage these cross-cutting concerns.

Although it is generally agreed that separating concerns is good software engineering practice, it is harder to pin down what is actually meant by a concern. Sometimes it is defined as a functional notion (i.e., a concern is some element of functionality in a system). Alternatively, it may be defined very broadly as ‘any piece of interest or

focus in a program'. Neither of these definitions is particularly useful in practice. Concerns certainly are more than simply functional elements but the more general definition is so vague that it is practically useless.

In my view, most attempts to define concerns are problematic because they attempt to relate concerns to programs. In fact, as discussed by Jacobson and Ng (2004), concerns are really reflections of the system requirements and priorities of stakeholders in the system. System performance may be a concern because users want to have a rapid response from a system; some stakeholders may be concerned that the system should include particular functionality; companies who are supporting a system may be concerned that it is easy to maintain. A concern can therefore be defined as something that is of interest or significance to a stakeholder or a group of stakeholders.

If you think of concerns as a way of organizing requirements, you can see why an approach to implementation that separates concerns into different program elements is good practice. It is easier to trace concerns, expressed as a requirement or a related set of requirements, to the program components that implement these concerns. If the requirements change, then the part of the program that has to be changed is obvious.

There are several different types of stakeholder concern:

1. Functional concerns, which are related to the specific functionality to be included in a system. For example, in a train control system, a specific functional concern is train braking.
2. Quality of service concerns, which are related to the non-functional behavior of a system. These include characteristics such as performance, reliability, and availability.
3. Policy concerns, which are related to the overall policies that govern the use of a system. Policy concerns include security and safety concerns and concerns related to business rules.
4. System concerns, which are related to attributes of the system as a whole, such as its maintainability or its configurability.
5. Organizational concerns, which are related to organizational goals and priorities. These include producing a system within budget, making use of existing software assets, and maintaining the reputation of the organization.

The core concerns of a system are those functional concerns that relate to its primary purpose. Therefore, for a hospital patient information system, the core functional concerns are the creation, editing, retrieval, and management of patient records. In addition to core concerns, large systems also have secondary functional concerns. These may involve functionality that shares information with the core concerns, or which is required so that the system can satisfy its non-functional requirements.

For example, consider a system that has a requirement to provide concurrent access to a shared buffer. One process adds data to the buffer and another process

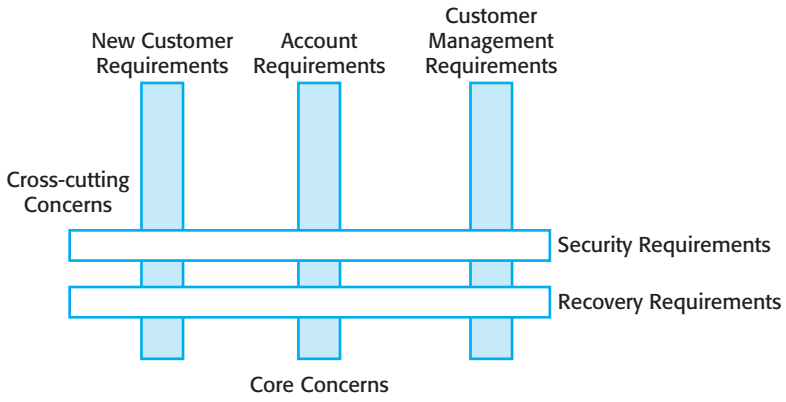

Figure 21.1 Cross-cutting concerns

takes data from the same buffer. This shared buffer is part of a data acquisition system where a producer process puts data in the buffer and a consumer process takes data from the buffer. The core concern here is to maintain a shared buffer so the core functionality is associated with adding and removing elements from the buffer. However, to ensure that the producer and consumer processes do not interfere with each other, there is an essential secondary concern of synchronization. The system must be designed so that the producer process cannot overwrite data that has not been consumed and the consumer process cannot take data from an empty buffer.

In addition to these secondary concerns, other concerns such as quality of service and organizational policies reflect essential system requirements. In general, these are system concerns—they apply to the system as a whole rather than to individual requirements or to the realization of these requirements in a program. These are called cross-cutting concerns to distinguish them from core concerns. Secondary functional concerns may also be cross-cutting although they do not always cross-cut the entire system; rather, they are associated with groupings of core concerns that provide related functionality.

Cross-cutting concerns are shown in Figure 21.1, which is based on an example of an Internet banking system. This system has requirements relating to new customers such as credit checking and address verification. It also has requirements related to the management of existing customers and the management of customer accounts. All of these are core concerns that are associated with the system's primary purpose—the provision of an Internet banking service. However, the system also has security requirements based on the bank's security policy, and recovery requirements to ensure that data is not lost in the event of a system failure. These are cross-cutting concerns as they may influence the implementation of all of the other system requirements.

Programming language abstractions, such as procedures and classes, are the mechanism that you normally use to organize and structure the core concerns of a system. However, the implementation of the core concerns in conventional programming languages usually includes additional code to implement the cross-cutting, functional, quality of service, and policy concerns. This leads to two undesirable phenomena: tangling and scattering.

```
synchronized void put (SensorRecord rec )
{
   // Check that there is space in the buffer; wait if not
   if ( numberOfEntries == bufsize)
      wait () ;
   // Add record at end of buffer
   store [back] = new SensorRecord (rec.sensorId, rec.sensorVal) ;
   back = back + 1 ;
   // If at end of buffer, next entry is at the beginning
   if (back == bufsize)
      back = 0 ;
   numberOfEntries = numberOfEntries + 1 ;
   // indicate that buffer is available
   notify () ;
} // put
```

Figure 21.2 Tangling of buffer management and synchronization code

Tangling occurs when a module in a system includes code that implements different system requirements. The example in Figure 21.2, which is a simplified implementation of part of the code for a bounded buffer system, illustrates this phenomenon. Figure 21.2 is an implementation of the put operation that adds an item for the buffer. However, if the buffer is full, it has to wait until a corresponding get operation removes an item from the buffer. The details are unimportant; essentially the wait () and notify () calls are used to synchronize the put and get operations. The code supporting the primary concern (in this case, putting a record into the buffer), is tangled with code implementing synchronization. Synchronization code, which is associated with the secondary concern of ensuring mutual exclusion, has to be included in all methods that access the shared buffer. Code associated with the synchronization concern is shown as shaded code in Figure 21.2.

The related phenomenon of scattering occurs when the implementation of a single concern (a logical requirement or set of requirements) is scattered across several components in a program. This is likely to occur when requirements related to secondary functional concerns or policy concerns are implemented.

For example, say a medical record management system, such as the MHC-PMS, has a number of components concerned with managing personal information, medication, consultations, medical images, diagnoses, and treatments. These implement the core concern of the system: maintaining records of patients. The system can be configured for different types of clinic by selecting the components that provide the functionality needed for the clinic.

However, assume there is also an important secondary concern which is the maintenance of statistical information; the health code provider wishes to record details of how many patients were admitted and discharged each month, how many patients died, what medications were issued, the reasons for consultations, and so on. These requirements have to be implemented by adding code that anonymizes the data (to maintain patient privacy) and writes it to a statistical database. A statistics component processes the statistical data and generates the statistic reports that are required.

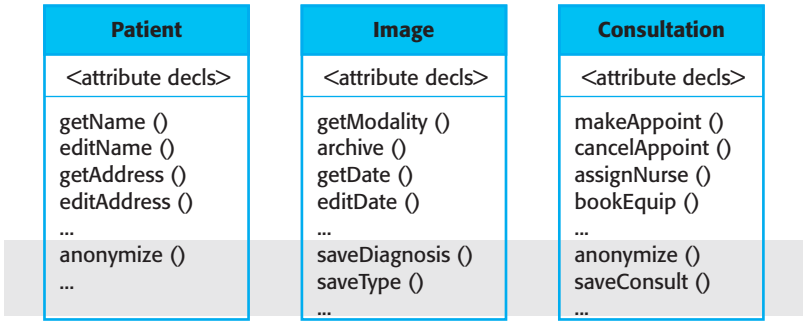

Figure 21.3 Scattering of methods implementing secondary concerns

This is illustrated in Figure 21.3. This diagram shows examples of three classes that might be included in the patient record system along with some of the core methods for managing patient information. The shaded area shows the methods that are required to implement the secondary statistics concern. You can see that this statistics concern is scattered throughout the other core concerns.

Problems with scattering and tangling occur when the initial system requirements change. For example, say new statistical data had to be collected in the patient record system. The changes to the system are not all located in one place and so you have to spend time looking for the components in the system that have to be changed. You then have to change each of these components to incorporate the required changes. This may be expensive because of the time required to analyze the components and then make and test the changes. There is always the possibility that you will miss some code that should be changed and so the statistics will be incorrect. Furthermore, as several changes have to be made, this increases the chances that you will make a mistake and introduce errors into the software.

21.2 Aspects, join points, and pointcuts

In this section, I introduce the most important new concepts associated with aspect-oriented software development and illustrate these using examples from the MHC-PMS. The terminology that I use was introduced by the developers of AspectJ in the late 1990s. However, the concepts are generally applicable and not specific to the AspectJ programming language. Figure 21.4 summarizes the key terms that you need to understand.

A medical records system such as the MHC-PMS includes components that handle logically related patient information. The patient component maintains personal information about a patient, the medication component holds information about medications that may be prescribed, and so on. By designing the system using a component-based approach, different instantiations of the system can be configured. For example, a version could be configured for each type of clinic with doctors only allowed to prescribe

Term	Definition
advice	The code implementing a concern.
aspect	A program abstraction that defines a cross-cutting concern. It includes the definition of a pointcut and the advice associated with that concern.
join point	An event in an executing program where the advice associated with an aspect may be executed.
join point model	The set of events that may be referenced in a pointcut.
pointcut	A statement, included in an aspect, that defines the join points where the associated aspect advice should be executed.
weaving	The incorporation of advice code at the specified join points by an aspect weaver.

Figure 21.4 Terminology used in aspect-oriented software engineering

medication relevant to that clinic. This simplifies the job of clinical staff and reduces the chances that a doctor will mistakenly prescribe the wrong medication.

However, this organization means that information in the database has to be updated from a number of different places in the system. For example, patient information may be modified when their personal details change, when their assigned medication changes, when they are assigned to a new specialist, etc. For simplicity, assume that all components in the system use a consistent naming strategy and that all database updates are implemented by methods starting with 'update'. There are therefore methods in the system such as:

```
updatePersonalInformation (patientId, infoupdate)
updateMedication (patientId, medicationupdate)
```

The patient is identified by patientId and the changes to be made are encoded in the second parameter; the details of this encoding are not important for this example. Updates are made by hospital staff, who are logged into the system.

Imagine that a security breach occurs in which patient information is maliciously changed. Perhaps someone has accidentally left his or her computer logged on and an unauthorized person has gained access to the system. Alternatively, an authorized insider may have gained access and maliciously changed the patient information. To reduce the probability of this happening again, a new security policy is introduced. Before any change to the patient database is made, the person requesting the change must reauthenticate himself or herself to the system. Details of who made the change are also logged in a separate file. This helps trace problems if they reoccur.

One way of implementing this new policy is to modify the update method in each component to call other methods to do the authentication and logging. Alternatively,

```
aspect authentication
{
   before: call (public void update* (..)) // this is a pointcut
   {
      // this is the advice that should be executed when woven into // the
      executing system
      int tries = 0 ;
      string userPassword = Password.Get ( tries ) ;
      while (tries < 3 && userPassword != thisUser.password ( ) )
      {
         // allow 3 tries to get the password right
         tries = tries + 1 ;
         userPassword = Password.Get ( tries ) ;
      }
      if (userPassword != thisUser.password ( )) then
         //if password wrong, assume user has forgotten to logout
         System.Logout (thisUser.uid) ;
      }
} // authentication
```

Figure 21.5 An authentication aspect

the system could be modified so that each time an update method is called, method calls are added before the call to do the authentication, and then after to log the changes made. However, neither of these is a very good solution to this problem:

1. The first approach leads to a tangled implementation. Logically, updating a database, authenticating the originator of an update, and logging details of the update are separate, unrelated concerns. You may wish to include authentication elsewhere in the system without logging or may wish to log actions apart from the update action. The same authentication and logging code has to be included within several different methods.
2. The alternative approach leads to a scattered implementation. If you explicitly include method calls to do authentication and logging before and after every call to the update methods, then this code is included at several different places in the system.

Authentication and logging cut across the core concerns of the system and may have to be included in several different places. In an aspect-oriented system, you can represent these cross-cutting concerns as separate aspects. An aspect includes a specification of where the cross-cutting concern is to be woven into the program, and code to implement that concern. This is illustrated in Figure 21.5, which defines an authentication aspect. The notation that I use in this example follows the style of AspectJ but uses a simplified syntax, which should be understandable without knowledge of either Java or AspectJ.

Aspects are completely different from other program abstractions in that the aspect itself includes a specification of where it should be executed. With other

abstractions, such as methods, there is a clear separation between the definition of the abstraction and its use. You cannot tell by examining the method where it will be called from; calls can be from anywhere that the method is in scope. Aspects, by contrast, include a 'pointcut'—a statement that defines where the aspect will be woven into the program.

In this example, the pointcut is a simple statement:

```
before: call (public void update* (..))
```

The meaning of this is that before the execution of any method whose name starts with the string update, followed by any other sequence of characters, the code in the aspect after the pointcut definition should be executed. The character * is called a wildcard and matches any string characters that are allowed in identifiers. The code to be executed is known as the 'advice' and is the implementation of the cross-cutting concern. In this case, the advice gets a password from the person requesting the change and checks that it matches the password of the currently logged-in user. If not, the user is logged out and the update does not proceed.

The ability to specify, using pointcuts, where code should be executed is the distinguishing characteristic of aspects. However, to understand what pointcuts mean, you need to understand another concept—the idea of a join point. A join point is an event that occurs during the execution of a program; so, it could be a method call, the initialization of a variable, the updating of a field, etc.

There are many possible types of event that may occur during program execution. A join point model defines the set of events that can be referenced in an aspect-oriented program. Join point models are not standardized and each aspect-oriented programming language has its own join point model. For example, in AspectJ events that are part of the join point model include:

- call events—calls to a method or a constructor;
- execution events—the execution of a method or a constructor;
- initialization events—class or object initialization;
- data events—accessing or updating of a field;
- exception events—the handling of an exception.

A pointcut identifies the specific event(s) (e.g., a call to a named procedure) with which advice should be associated. This means that you can weave advice into a program in many different contexts, depending on the join point model that is supported:

1. Advice can be included before the execution of a specific method, a list of named methods, or a list of methods whose names match a pattern specification (such as update*).

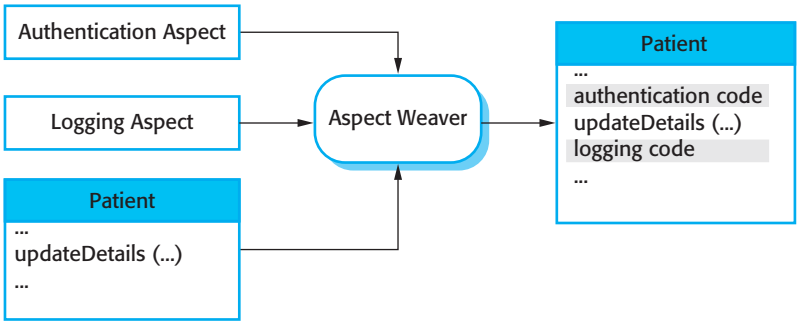

Figure 21.6 Aspect weaving

2. Advice can be included after the normal or exceptional return from a method. In the example shown in Figure 21.5, you could define a pointcut that would execute the logging code after all calls to update methods.
3. Advice can be included when a field in an object is modified; you can include advice to monitor or change that field.

The inclusion of advice at the join points specified in the pointcuts is the responsibility of an aspect weaver. Aspect weavers are extensions to compilers that process the definition of aspects and the object classes and methods that define the system. The weaver then generates a new program with the aspects included at the specified join points. The aspects are integrated so that the cross-cutting concerns are executed at the right places in the final system.

Figure 21.6 illustrates this aspect weaving for the authentication and logging aspects that should be included in the MHC-PMS. There are three different approaches to aspect weaving:

1. Source code pre-processing, where a weaver takes source code input and generates new source code in a language such as Java or C++, which can then be compiled using the standard language compiler. This approach has been adopted for the AspectX language with its associated XWeaver (Birrer et al., 2005).
2. Link time weaving, where the compiler is modified to include an aspect weaver. An aspect-oriented language such as AspectJ is processed and standard Java bytecode is generated. This can then be executed directly by a Java interpreter or further processed to generate native machine code.
3. Dynamic weaving at execution time. In this case, join points are monitored and when an event that is referenced in a pointcut occurs, the corresponding advice is integrated with the executing program.

The most commonly used approach to aspect weaving is link time weaving, as this allows for the efficient implementation of aspects without a large run-time overhead. Dynamic weaving is the most flexible approach but can incur significant performance penalties during program execution. Source code pre-processing is now rarely used.

21.3 Software engineering with aspects

Aspects were originally introduced as a programming language construct but, as I have discussed, the notion of concerns is one that really comes from the system requirements. Therefore, it makes sense to adopt an aspect-oriented approach at all stages of the system development process. In the early stages of software engineering, adopting an aspect-oriented approach means using the notion of separating concerns as a basis for thinking about the requirements and the system design. Identifying and modeling concerns should be part of the requirements engineering and design processes. Aspect-oriented programming languages then provide the technological support to maintain the separation of concerns in your implementation of the system.

When designing a system, Jacobson and Ng (2004) suggest that you should think of a system that supports different stakeholder concerns as a core system plus extensions. I have illustrated this in Figure 21.7, where I have used UML packages to represent both the core and the extensions. The core system is a set of system features that implements the essential purpose of a system. Therefore, if the purpose of a particular system is to maintain information on patients in a hospital, then the core system provides a means of creating, editing, managing, and accessing a database of patient records. The extensions to the core system reflect additional stakeholder concerns, which must be integrated with the core system. For example, it is important that a medical information system maintains the confidentiality of patient information, so one extension might be concerned with access control, another with encryption, etc.

There are several different types of extension that are derived from the different types of concern that I discussed in Section 21.1.

1. *Secondary functional extensions* These add additional capabilities to the functionality provided in the core system. For instance, using the example of the MHC-PMS, the production of reports on the drugs prescribed in the previous month would be a secondary functional extension to a patient information system.
2. *Policy extensions* These add functional capabilities to support organizational policies. Extensions that add security features are examples of policy extensions.
3. *QoS extensions* These add functional capabilities to help attain the quality of service requirements that have been specified for the system. For example, an extension might implement a cache to reduce the number of database accesses or automated backups for recovery in the event of a system failure.
4. *Infrastructure extensions* These extensions add functional capabilities to support the implementation of a system on some specific implementation platform. For example, in a patient information system, infrastructure extensions might be used to implement the interface to the underlying database management system. Changes to this interface can be made by modifying the associated infrastructure extensions.

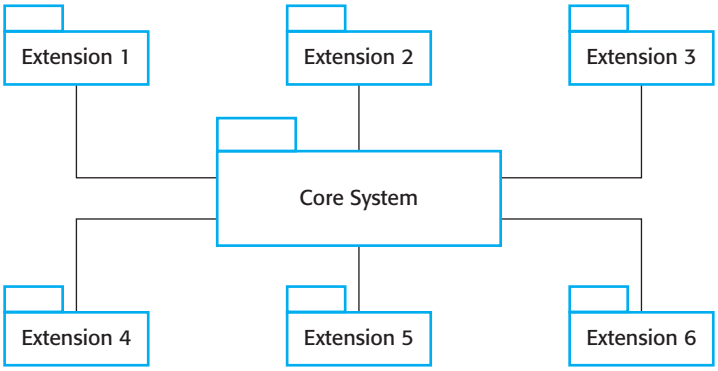

Figure 21.7 Core system with extensions

Extensions always add some kind of functionality or additional features to the core system. Aspects are a way to implement these extensions and they can be composed with the core system functionality using the weaving facilities in the aspect-oriented programming environment.

21.3.1 Concern-oriented requirements engineering

As I suggested in Section 21.1, concerns reflect the requirements of stakeholders. These concerns may reflect the functionality required by a stakeholder, the quality of system service, organizational policies or issues that are related to the attributes of the system as a whole. It therefore makes sense to adopt an approach to requirements engineering that identifies and specifies the different stakeholder concerns. The term 'early aspects' is sometimes used to refer to the use of aspects at early stages in the software lifecycle where the separation of concerns is emphasized.

The importance of separating concerns during requirements engineering has been recognized for many years. Viewpoints that represent different system perspectives have been incorporated into a number of requirements engineering methods (Easterbrook and Nuseibeh, 1996; Finkelstein et al., 1992; Kotonya and Sommerville, 1996). These methods separate the concerns of different stakeholders. Viewpoints reflect the distinct functionality that is required by different stakeholder groups.

However, there are also requirements which cross-cut all viewpoints, as shown in Figure 21.8. This diagram shows that viewpoints may be of different types but cross-cutting concerns (such as regulation, dependability, and security) generate requirements that may impact on all of the system viewpoints. This was the major consideration in the work which I did in the development of the PreView method (Sommerville and Sawyer, 1997; Sommerville et al., 1998), which included steps to identify cross-cutting, non-functional concerns.

To develop a system that is organized in the style shown in Figure 21.7, you should identify requirements for the core system plus the requirements for the system extensions. A viewpoint-oriented approach to requirements engineering, where each viewpoint represents the requirements of related groups of stakeholders, is one

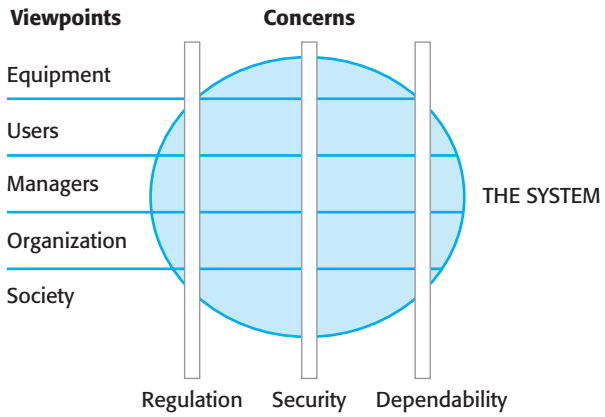

Figure 21.8 Viewpoints and Concerns

way to separate core and secondary concerns. If you organize the requirements according to stakeholder viewpoint, you can then analyze them to discover related requirements that appear in all or most viewpoints. These represent the core functionality of the system. Other viewpoint requirements may be requirements that are specific to that viewpoint. These can be implemented as extensions to the core functionality.

For example, imagine that you are developing a software system to keep track of specialized equipment used by the emergency services. Equipment is located at different places across a region or state and, in the event of an emergency such as a flood or earthquake, the emergency services use the system to discover what equipment is available close to the site of the problem. Figure 21.9 shows outline requirements from three possible viewpoints for such a system.

You can see from this example that stakeholders from all of the different viewpoints need to be able to find specific items of equipment, browse the equipment available at each location, and check in/check out equipment from the store. These are therefore requirements for the core system. The secondary requirements support the more specific needs of each viewpoint. There are secondary requirements for system extensions supporting equipment use, management, and maintenance.

The secondary functional requirements that are identified from any one viewpoint do not, necessarily, cross-cut the requirements from other viewpoints. For example, only the maintenance viewpoint is interested in completing maintenance records. These requirements reflect the needs of that viewpoint and those concerns may not be shared with other viewpoints. In addition to the secondary functional requirements, however, there are cross-cutting concerns that generate requirements of importance to some or all viewpoints. These often reflect policy and quality of service requirements that apply to the system as a whole. As I discussed in Chapter 4, these are non-functional requirements such as requirements for security, performance, and cost.

In the equipment inventory system, an example of a cross-cutting concern is system availability. Emergencies may happen with little or no warning. Saving lives may require essential equipment to be deployed as quickly as possible. Therefore, the

1. Emergency service users

1.1 Find a specified type of equipment (e.g., heavy lifting gear)
1.2 View equipment available in a specified store
1.3 Check-out equipment
1.4 Check-in equipment
1.5 Arrange equipment to be transported to emergency
1.6 Submit damage report
1.7 Find store close to emergency

2. Emergency planners

2.1 Find a specified type of equipment
2.2 View equipment available in a specified location
2.3 Check in/check out equipment from a store
2.4 Move equipment from one store to another
2.6 Order new equipment

3. Maintenance staff

3.1 Check in/check out equipment for maintenance
3.2 View equipment available at each store
3.3 Find a specified type of equipment
3.4 View maintenance schedule for an equipment item
3.5 Complete maintenance record for an equipment item
3.6 Show all items in a store requiring maintenance

Figure 21.9 Viewpoints on an equipment inventory system

dependability requirements for the equipment inventory system include requirements for a high level of system availability. Some examples of these dependability requirements, with associated rationale, are shown in Figure 21.10. Using these requirements, you can then identify extensions to the core functionality for transaction logging and status reporting. These make it easier to identify problems and switch to a backup system.

The outcome of the requirements engineering process should be a set of requirements that are structured around the notion of a core system plus extensions. For example, in the inventory system, examples of core requirements might be:

C.1 The system shall allow authorized users to view the description of any item of equipment in the emergency services inventory.

Figure 21.10 Availability-related requirements for the equipment inventory system

AV.1	There shall be a 'hot standby' system available in a location that is geographically well-separated from the principal system.
Rationale:	The emergency may affect the principal location of the system.
AV.1.1	All transactions shall be logged at the site of the principal system and at the remote standby site.
Rationale:	This allows these transactions to be replayed and the system databases made consistent.
AV.1.2	The system shall send status information to the emergency control room system every five minutes.
Rationale:	The operators of the control room system can switch to the hot standby if the principal system is unavailable.

Viewpoints

I introduced the notion of viewpoints in Chapter 4, where I explained how viewpoints could be used as a way of structuring the requirements from different stakeholders. Using viewpoints, you can identify the requirements for the core system from each stakeholder grouping.

http://www.SoftwareEngineering-9.com/Web/Requirements/Viewpoints.html

C.2 The system shall include a search facility to allow authorized users to search either individual inventories or the complete inventory for a specific item of equipment or a specific type of equipment.

The system may also include an extension that is intended to support equipment procurement and replacement. Requirements for this extension might be:

E1.1 It shall be possible for authorized users to place orders with accredited suppliers for replacement items of equipment.

E1.1.1 When an item of equipment is ordered, it should be allocated to a specific inventory and flagged in that inventory as 'on order'.

As a general rule, you should avoid having too many concerns or extensions to the system. These simply confuse the reader and may lead to premature design. This limits the freedom of designers and may result in a system design that cannot meet its quality of service requirements.

21.3.2 Aspect-oriented design and programming

Aspect-oriented design is the process of designing a system that makes use of aspects to implement the cross-cutting concerns and extensions that are identified during the requirements engineering process. At this stage, you need to translate the concerns that relate to the problem to be solved to corresponding aspects in the program that is implementing the solution. You also need to understand how these aspects will be composed with other system components and ensure that composition ambiguities do not arise.

The high-level statement of requirements provides a basis for identifying some system extensions that may be implemented as aspects. You then need to develop these in more detail to identify further extensions and to understand the functionality that is required. One way to do this is to identify a set of use cases, (discussed in Chapters 4 and 5) associated with each viewpoint. Use case models are interaction-focused and more detailed than the user requirements. You can think of them as a bridge between the requirements and the design. In a use case model, you describe

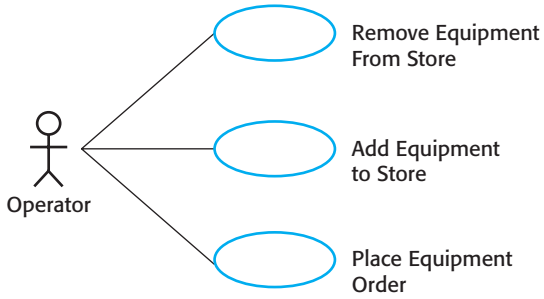

Figure 21.11 Use cases from the inventory management system

the steps of each user interaction and so start to identify and define the classes in the system.

Jacobson and Ng (2004) have written a book that discusses how use cases can be used in aspect-oriented software engineering. They suggest that each use case represents an aspect and propose extensions to the use case approach to support join points and pointcuts. They also introduce the notion of use case slices and use case modules. These include fragments of classes that implement an aspect. They can be composed to create the complete system.

Figure 21.11 shows examples of three use cases that might be part of the inventory management system. These reflect the concerns of adding equipment to an inventory and ordering equipment. Equipment ordering and adding equipment to a store are related concerns. Once ordered items have been delivered, they must be added to the inventory and delivered to one of the equipment stores.

The UML already includes the notion of extension use cases. An extension use case extends the functionality of another use case. Figure 21.12 shows how the placing of an equipment order extends the core use case for adding equipment to a specific store. If the equipment to be added does not exist, it can be ordered and added to the store when the equipment is delivered. During the development of use case models, you should look for common features and, where possible, structure the use cases as core cases plus extensions. Cross-cutting features, such as the logging of all transactions, can also be represented as extension use cases. Jacobsen and Ng discuss how extensions of this type can be implemented as aspects.

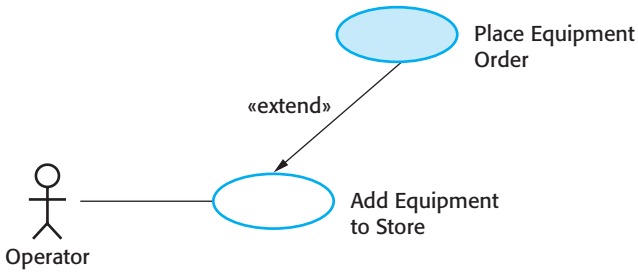

Figure 21.12 Extension use cases

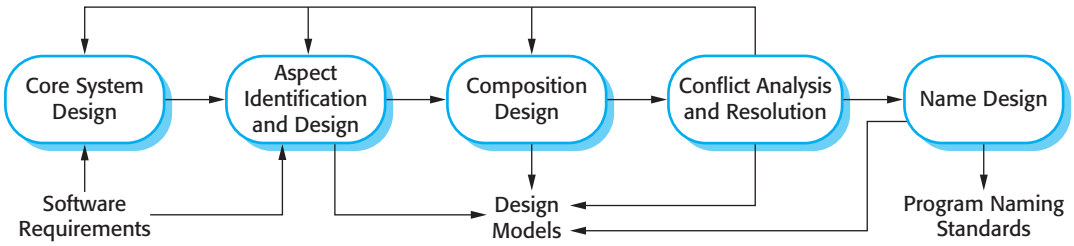

Figure 21.13 A generic aspect-oriented design process

Developing an effective process for aspect-oriented design is essential if aspect-oriented design is to be accepted and used. I suggest that an aspect-oriented design process should include the activities shown in Figure 21.13. These activities are:

1. *Core system design* At this stage, you design the system architecture to support the core functionality of the system. The architecture must also take into account quality of service requirements such as performance and dependability requirements.
2. *Aspect identification and design* Starting with the extensions identified in the system requirements, you should analyze these to see if they are aspects in themselves or if they should be broken down into several aspects. Once aspects have been identified, these can then be separately designed, taking into account the design of the core system features.
3. *Composition design* At this stage, you analyze the core system and aspect designs to discover where the aspects should be composed with the core system. Essentially, you are identifying the join points in a program at which aspects will be woven.
4. *Conflict analysis and resolution* A problem with aspects is that they may interfere with each other when they are composed with the core system. Conflicts occur when there is a pointcut clash with different aspects specifying that they should be composed at the same point in the program. However, there may be more subtle conflicts. When aspects are designed independently, they may make assumptions about the core system functionality that has to be modified. However, when several aspects are composed, one aspect may affect the functionality of the system in a way that was not anticipated by other aspects. The overall system behavior may then not be as expected.
5. *Name design* This is an important design activity that defines standards for naming entities in the program. This is essential to avoid the problem of accidental pointcuts. These occur when, at some program join point, the name accidentally matches that in a pointcut pattern. The advice is therefore unintentionally applied at that point. Obviously this is undesirable and can lead to unexpected program behavior. Therefore, you should design a naming scheme that minimizes the likelihood of this happening.

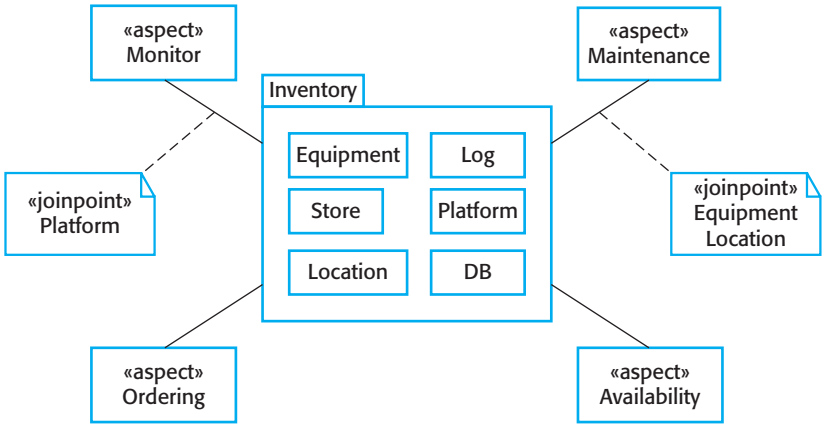

Figure 21.14 An aspect-oriented design model

This process is, naturally, an iterative process in which you make initial design proposals then refine them as you analyze and understand the design issues. Normally, you would expect to refine the extensions identified in the requirements to a larger number of aspects.

The outcome of the aspect-oriented design process is an aspect-oriented design model. This may be expressed in an extended version of the UML which includes new, aspect-specific constructs such as those proposed by Clarke and Baniassad (2005) and Jacobson and Ng (2004). The essential elements of 'aspect UML' are a means of modeling aspects and of specifying the join points at which the aspect advice should be composed with the core system.

Figure 21.14 is an example of an aspect-oriented design model. I have used the UML stereotype for an aspect proposed by Jacobson and Ng. Figure 21.14 shows the core system for an emergency services inventory plus some aspects that might be composed with that core. I have shown some core system classes and some aspects. This is a simplified picture; a complete model would include more classes and aspects. Notice how I have used UML notes to provide additional information about the classes that are cross-cut by some aspects.

Figure 21.15 is a more detailed model of an aspect. Obviously, before you design aspects, you have to have a core system design. As I don't have space to show this here, I have made a number of assumptions about classes and methods in the core system.

The first section of the aspect sets out the pointcuts that specify where it will be composed with the core system. For example, the first pointcut specifies that the aspect may be composed at the call getItemInfo (..) join point. The following section defines the extensions that are implemented by the aspect. In the example here, the extension statement can be read as:

> *"In the method viewItem, after the call to the method getItemInfo, a call to the method displayHistory should be included to display the maintenance record."*

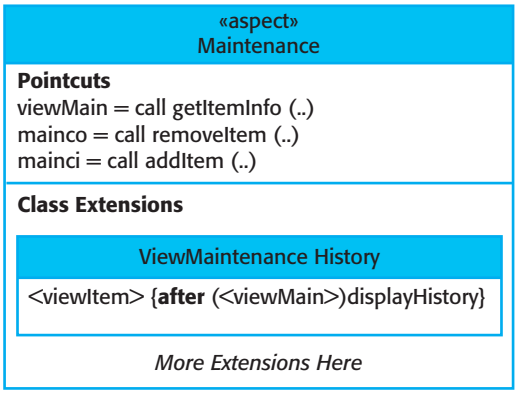

Figure 21.15 Part of a model of an aspect

Aspect-oriented programming (AOP) started at Xerox's PARC laboratories in 1997, with the development of the AspectJ programming language. This remains the most widely used aspect-oriented language, although aspect-oriented extensions of other languages, such as C# and C++, have also been implemented. Other experimental languages have also been developed to support the explicit separation of concerns and concern composition and there are experimental implementation of AOP in the .NET framework. Aspect-oriented programming is covered extensively in other books (Colyer et al., 2005; Gradecki and Lezeiki, 2003; Laddad, 2003b).

If you have followed an aspect-oriented approach to designing your system, you will already have identified the core functionality and the extensions to that functionality to be implemented as cross-cutting aspects. The focus of the programming process should then be to write code implementing the core and extension functionality and, critically, to specify the pointcuts in the aspects so that the aspect advice is woven into the base code at the correct places.

Correctly specifying pointcuts is very important as these define where the aspect advice will be composed with the core functionality. If you make a mistake in pointcut specification, then the aspect advice will be woven into the program in the wrong place. This could lead to unexpected and unpredictable program behavior. Adherence to the naming standards established during system design is essential. You also have to review all of the aspects to ensure that aspect interference will not occur if two or more aspects are woven into the core system at the same join point. In general, it is best to avoid this completely but, occasionally, it might be the best way to implement a concern. In those circumstances, you have to ensure that the aspects are completely independent. The program's behavior should not depend on the order that the aspects are woven into the program.

21.3.3 Verification and validation

As I discussed in Chapter 8, verification and validation is the process of demonstrating that a program meets its specification (verification) and meets the real needs of its stakeholders (validation). Static verification techniques focus on manual or automated

analysis of the source code of the program. Dynamic validation or testing is used to discover defects in the program or to demonstrate that the program meets its requirements. When defect detection is the objective, the testing process may be guided by knowledge of the program's source code. Test coverage metrics show the effectiveness of tests in causing source code statements to be executed.

For aspect-oriented systems, the processes of validation testing are no different than for any other system. The final executable program is treated as a black box and tests are devised to show whether or not the system meets its requirements. However, the use of aspects causes real problems with program inspections and white-box testing, where the program source code is used to identify potential defect tests.

Program inspections, which I describe in Chapter 24, involve a team of readers looking at the source code of a program to discover defects that have been introduced by the programmer. It is a very effective technique of defect discovery. However, aspect-oriented programs cannot be read sequentially (i.e., from top to bottom). They are therefore more difficult for people to understand.

A general guideline for program understandability is that a reader should be able to read a program from left to right, top to bottom without having to switch attention to other parts of the code. This makes it easier for readers and also makes it less likely that programmers will make mistakes as their attention is focused on a single section of code. Improving program readability was a key reason for the introduction of structured programming (Dijkstra et al., 1972) and the elimination of unconditional branch (go-to) statements from high-level programming languages.

In an aspect-oriented system, sequential code reading is impossible. The reader has to examine each aspect, understand its pointcuts (which may be patterns). and the join point model of the aspect-oriented language. When reading the program, he or she then has to identify every potential join point and switch attention to the aspect code to see if it may be woven at that point. Their attention then returns to the main flow of control of the base code. In reality, this is cognitively impossible and the only possible way to inspect an aspect-oriented program is through the use of code-reading tools.

Code-reading tools can be written that 'flatten' an aspect-oriented program and present a program to the reader with the aspects 'woven' into the program at the specified join points. However, this is not a complete solution to the code-reading problem. The join point model in an aspect-oriented programming language may be dynamic rather than static and it may be impossible to demonstrate that the flattened program will behave in exactly the same way as the program that will execute. Furthermore, because it is possible for different aspects to have the same pointcut specification, the program-reading tool must know how the aspect weaver handles these 'competing' aspects and how the composition will be ordered.

White-box or structural testing is a systematic approach to testing where knowledge of the program source code is used to design defect tests. The aim is to design tests that provide some level of program coverage. That is, the set of tests should ensure that every logical path through the program is executed, with the consequence that each program statement is executed at least once. Program execution analyzers may be used to demonstrate that this level of test coverage has been achieved.

In an aspect-oriented system, there are two problems with this approach:

1. How can knowledge of the program code be used to systematically derive program tests?
2. What exactly does test coverage mean?

To design tests in a structured program (e.g., tests of the code of a method) without unconditional branches, you can derive a program flow graph, which reveals every logical execution path through that program. You then examine the code and, for each path through the flow graph, choose input values that will cause that path to be executed.

However, an aspect-oriented program is not a structured program. The flow of control is interrupted by 'come from' statements (Constantinos et al., 2004). At some join point in the execution of the base code, an aspect may be executed. I am not sure that it is possible to construct a structured flow diagram in such a situation. It is therefore difficult to systematically design program tests that ensure that all combinations of base code and aspects are executed.

In an aspect-oriented program, there is also the problem of deciding what 'test coverage' means. Does it mean that the code of each aspect is executed at least once? This is a very weak condition because of the interaction between aspects and the base code at the join points where the aspects are woven. Should the idea of test coverage be extended so that the code of the aspect is executed at least once at every join point specified in the aspect pointcut? In such situations, what happens if different aspects define the same pointcut? These are both theoretical and practical problems. We need tools to support aspect-oriented program testing which will help assess the extent of test coverage of a system.

As I discuss in Chapter 24, large projects normally have a separate quality assurance team who set testing standards and who require a formal assurance that program reviews and testing have been completed to these standards. The problems of inspecting and deriving tests for aspect-oriented programs are a significant barrier to the adoption of aspect-oriented software development in such large software projects.

As well as problems with inspections and white-box testing, Katz (2005) identified additional problems in testing aspect-oriented programs:

1. How should aspects be specified so that tests for these aspects may be derived?
2. How can aspects be tested independently of the base system with which they should be woven?
3. How can aspect interference be tested? As I have discussed, aspect interference occurs when two or more aspects use the same pointcut specification.
4. How can tests be designed so that all program join points are executed and appropriate aspect tests applied?

Fundamentally, these testing problems occur because aspects are tightly rather than loosely integrated with the base code of a system. They are therefore difficult to test in isolation. Because they may be woven into a program in many different places, you can't be sure that an aspect that works successfully at one join point will necessarily work at all join points. All of these remain research problems for aspect-oriented software development.

KEY POINTS

- The main benefit of an aspect-oriented approach to software development is that it supports the separation of concerns. By representing cross-cutting concerns as aspects, individual concerns can be understood, reused, and modified without changing other parts of the program.
- Tangling occurs when a module in a system includes code that implements different system requirements. The related phenomenon of scattering occurs when the implementation of a single concern is scattered across several components in a program.
- Aspects include a pointcut—a statement that defines where the aspect will be woven into the program, and advice—the code to implement the cross-cutting concern. Join points are the events that can be referenced in a pointcut.
- To ensure the separation of concerns, systems can be designed as a core system that implements the primary concerns of stakeholders, and a set of extensions that implement secondary concerns.
- To identify concerns, you may use a viewpoint-oriented approach to requirements engineering to elicit stakeholder requirements and to identify cross-cutting quality of service and policy concerns.
- The transition from requirements to design can be made by identifying use cases, where each use case represents a stakeholder concern. The design may be modeled using an extended version of the UML with aspect stereotypes.
- The problems of inspecting and deriving tests for aspect-oriented programs are a significant barrier to the adoption of aspect-oriented software development in large software projects.

FURTHER READING

'Aspect-oriented programming'. This special issue of the CACM has a number of articles for a general audience, which are a good starting point for reading about aspect-oriented programming (*Comm. ACM*, **44** (10), October 2001.) http://dx.doi.org/10.1145/383845.383846.

Aspect-oriented Software Development. A multiauthor book with a wide range of papers on aspect-oriented software development, written by many of the leading researchers in the field. (R. E. Filman, T. Elrad, S. Clarke and M. Aksit, Addison-Wesley, 2005.)

Aspect-oriented Software Development with Use cases. This is a practical book for software designers. The authors discuss how to use use cases to manage the separation of concerns, and to use these as the basis of an aspect-oriented design. (I. Jacobson and P. Ng, Addison-Wesley, 2005.)

EXERCISES

21.1. What are the different types of stakeholder concern that may arise in a large system? How can aspects support the implementation of each of these types of concern?

21.2. Summarize what is meant by tangling and scattering. Using examples, explain why tangling and scattering can cause problems when system requirements change.

21.3. What is the difference between a join point and a pointcut? Explain how these facilitate the weaving of code into a program to handle cross-cutting concerns.

21.4. What assumptions underlie the idea that a system should be organized as a core system that implements the essential requirements, plus extensions that implement additional functionality? Can you think of systems where this model would not be appropriate?

21.5. What viewpoints should be considered when developing a requirements specification for the MHC-PMS? What are likely to be the most important cross-cutting concerns?

21.6. Using the outline functionality for each viewpoint shown in Figure 21.9, identify six further use cases for the equipment inventory system, in addition to those shown in Figure 21.11. Where appropriate, show how some of these might be organized as extension use cases.

21.7. Using the aspect stereotype notation illustrated in Figure 21.15, develop in more detail the Ordering and Monitor aspects, shown in Figure 21.14.

21.8. Explain how aspect interference can arise and suggest what should be done during the system design process to reduce the problems of aspect interference.

21.9. Explain why expressing pointcut specifications as patterns increases the problems of testing aspect-oriented programs. To answer this, think about how program testing normally involves comparing the expected output to the actual output produced by a program.

21.10. Suggest how you could use aspects to simplify the debugging of programs.