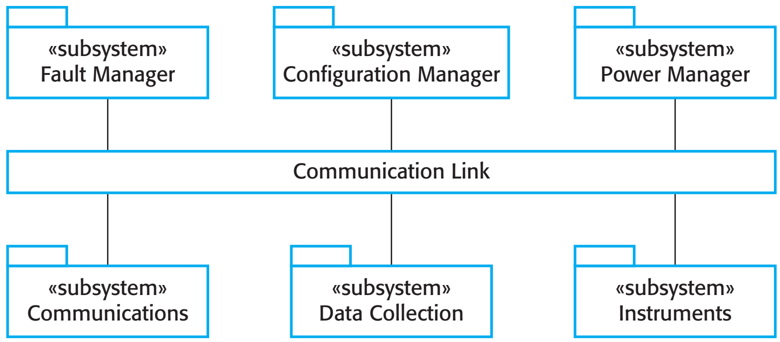



**Figure 7.4** High-level architecture of the weather station

You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client–server model. However, this is not essential at this stage.

The high-level architectural design for the weather station software is shown in Figure 7.4. The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure, shown as the Communication link in Figure 7.4. Each subsystem listens for messages on that infrastructure and picks up the messages that are intended for them. This is another commonly used architectural style in addition to those described in Chapter 6.

For example, when the communications subsystem receives a control command, such as shutdown, the command is picked up by each of the other subsystems, which then shut themselves down in the correct way. The key benefit of this architecture is that it is easy to support different configurations of subsystems because the sender of a message does not need to address the message to a particular subsystem.

Figure 7.5 shows the architecture of the data collection subsystem, which is included in Figure 7.4. The Transmitter and Receiver objects are concerned with managing communications and the WeatherData object encapsulates the information that is collected from the instruments and transmitted to the weather information system. This arrangement follows the producer-consumer pattern, discussed in Chapter 20.

## 7.1.3 Object class identification

By this stage in the design process, you should have some ideas about the essential objects in the system that you are designing. As your understanding of the design develops, you refine these ideas about the system objects. The use case description helps to identify objects and operations in the system. From the description of the Report weather use case, it is obvious that objects representing the instruments that collect weather data will be required, as will an object representing the summary of the weather data. You also usually need a high-level

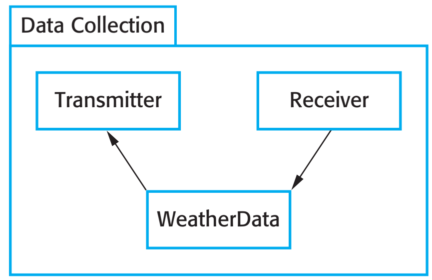



**Figure 7.5** Architecture of data collection system

system object or objects that encapsulate the system interactions defined in the use cases. With these objects in mind, you can start to identify the object classes in the system.

There have been various proposals made about how to identify object classes in object-oriented systems:

1. Use a grammatical analysis of a natural language description of the system to be constructed. Objects and attributes are nouns; operations or services are verbs (Abbott, 1983).

2. Use tangible entities (things) in the application domain such as aircraft, roles such as manager or doctor, events such as requests, interactions such as meetings, locations such as offices, organizational units such as companies, and so on (Coad and Yourdon, 1990; Shlaer and Mellor, 1988; Wirfs-Brock et al., 1990).

3. Use a scenario-based analysis where various scenarios of system use are identified and analyzed in turn. As each scenario is analyzed, the team responsible for the analysis must identify the required objects, attributes, and operations (Beck and Cunningham, 1989).

In practice, you have to use several knowledge sources to discover object classes. Object classes, attributes, and operations that are initially identified from the informal system description can be a starting point for the design. Further information from application domain knowledge or scenario analysis may then be used to refine and extend the initial objects. This information can be collected from requirements documents, discussions with users, or from analyses of existing systems.

In the wilderness weather station, object identification is based on the tangible hardware in the system. I don't have space to include all the system objects here, but I have shown five object classes in Figure 7.6. The Ground thermometer, Anemometer, and Barometer objects are application domain objects, and the WeatherStation and WeatherData objects have been identified from the system description and the scenario (use case) description:

1. The WeatherStation object class provides the basic interface of the weather station with its environment. Its operations reflect the interactions shown in

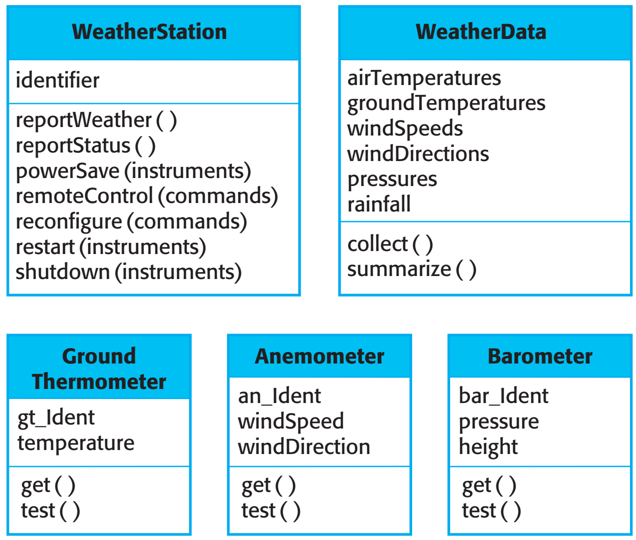



**Figure 7.6** Weather station objects

Figure 7.3. In this case, I use a single object class to encapsulate all of these interactions, but in other designs you could design the system interface as several different classes.

2. The WeatherData object class is responsible for processing the report weather command. It sends the summarized data from the weather station instruments to the weather information system.

3. The Ground thermometer, Anemometer, and Barometer object classes are directly related to instruments in the system. They reflect tangible hardware entities in the system and the operations are concerned with controlling that hardware. These objects operate autonomously to collect data at the specified frequency and store the collected data locally. This data is delivered to the WeatherData object on request.

You use knowledge of the application domain to identify other objects, attributes, and services. We know that weather stations are often located in remote places and include various instruments that sometimes go wrong. Instrument failures should be reported automatically. This implies that you need attributes and operations to check the correct functioning of the instruments. There are many remote weather stations so each weather station should have its own identifier.

At this stage in the design process, you should focus on the objects themselves, without thinking about how these might be implemented. Once you have identified the objects, you then refine the object design. You look for common features and then design the inheritance hierarchy for the system. For example, you may identify an Instrument superclass, which defines the common features of all instruments, such as an identifier, and get and test operations. You may also add new attributes and operations to the superclass, such as an attribute that maintains the frequency of data collection.

### 7.1.4 Design models

Design or system models, as I discussed in Chapter 5, show the objects or object classes in a system. They also show the associations and relationships between these entities. These models are the bridge between the system requirements and the implementation of a system. They have to be abstract so that unnecessary detail doesn't hide the relationships between them and the system requirements. However, they also have to include enough detail for programmers to make implementation decisions.

Generally, you get around this type of conflict by developing models at different levels of detail. Where there are close links between requirements engineers, designers, and programmers, then abstract models may be all that are required. Specific design decisions may be made as the system is implemented, with problems resolved through informal discussions. When the links between system specifiers, designers, and programmers are indirect (e.g., where a system is being designed in one part of an organization but implemented elsewhere), then more detailed models are likely to be needed.

An important step in the design process, therefore, is to decide on the design models that you need and the level of detail required in these models. This depends on the type of system that is being developed. You design a sequential data-processing system in a different way from an embedded real-time system, so you will need different design models. The UML supports 13 different types of models but, as I discussed in Chapter 5, you rarely use all of these. Minimizing the number of models that are produced reduces the costs of the design and the time required to complete the design process.

When you use the UML to develop a design, you will normally develop two kinds of design model:

1. Structural models, which describe the static structure of the system using object classes and their relationships. Important relationships that may be documented at this stage are generalization (inheritance) relationships, uses/used-by relationships, and composition relationships.

2. Dynamic models, which describe the dynamic structure of the system and show the interactions between the system objects. Interactions that may be documented include the sequence of service requests made by objects and the state changes that are triggered by these object interactions.

In the early stages of the design process, I think there are three models that are particularly useful for adding detail to use case and architectural models:

1. Subsystem models, which that show logical groupings of objects into coherent subsystems. These are represented using a form of class diagram with each subsystem shown as a package with enclosed objects. Subsystem models are static (structural) models.

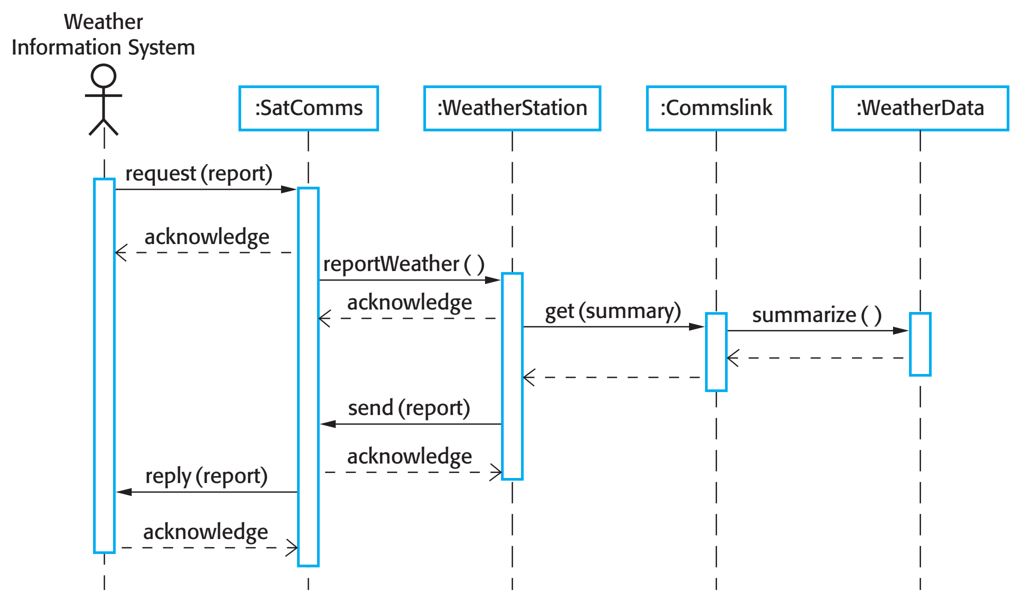



**Figure 7.7** Sequence diagram describing data collection

2. Sequence models, which show the sequence of object interactions. These are represented using a UML sequence or a collaboration diagram. Sequence models are dynamic models.

3. State machine model, which show how individual objects change their state in response to events. These are represented in the UML using state diagrams. State machine models are dynamic models.

A subsystem model is a useful static model as it shows how a design is organized into logically related groups of objects. I have already shown this type of model in Figure 7.4 to show the subsystems in the weather mapping system. As well as subsystem models, you may also design detailed object models, showing all of the objects in the systems and their associations (inheritance, generalization, aggregation, etc.). However, there is a danger in doing too much modeling. You should not make detailed decisions about the implementation that really should be left to the system programmers.

Sequence models are dynamic models that describe, for each mode of interaction, the sequence of object interactions that take place. When documenting a design, you should produce a sequence model for each significant interaction. If you have developed a use case model then there should be a sequence model for each use case that you have identified.

Figure 7.7 is an example of a sequence model, shown as a UML sequence diagram. This diagram shows the sequence of interactions that take place when an external system requests the summarized data from the weather station. You read sequence diagrams from top to bottom:

1. The SatComms object receives a request from the weather information system to collect a weather report from a weather station. It acknowledges receipt of

this request. The stick arrowhead on the sent message indicates that the external system does not wait for a reply but can carry on with other processing.

2. SatComms sends a message to WeatherStation, via a satellite link, to create a summary of the collected weather data. Again, the stick arrowhead indicates that SatComms does not suspend itself waiting for a reply.

3. WeatherStation sends a message to a Commslink object to summarize the weather data. In this case, the squared-off style of arrowhead indicates that the instance of the WeatherStation object class waits for a reply.

4. Commslink calls the summarize method in the object WeatherData and waits for a reply.

5. The weather data summary is computed and returned to WeatherStation via the Commslink object.

6. WeatherStation then calls the SatComms object to transmit the summarized data to the weather information system, through the satellite communications system.

The SatComms and WeatherStation objects may be implemented as concurrent processes, whose execution can be suspended and resumed. The SatComms object instance listens for messages from the external system, decodes these messages and initiates weather station operations.

Sequence diagrams are used to model the combined behavior of a group of objects but you may also want to summarize the behavior of an object or a subsystem in response to messages and events. To do this, you can use a state machine model that shows how the object instance changes state depending on the messages that it receives. The UML includes state diagrams, initially invented by Harel (1987) to describe state machine models.

Figure 7.8 is a state diagram for the weather station system that shows how it responds to requests for various services.

You can read this diagram as follows:

1. If the system state is Shutdown then it can respond to a restart(), a reconfigure(), or a powerSave() message. The unlabeled arrow with the black blob indicates that the Shutdown state is the initial state. A restart() message causes a transition to normal operation. Both the powerSave() and reconfigure() messages cause a transition to a state in which the system reconfigures itself. The state diagram shows that reconfiguration is only allowed if the system has been shut down.

2. In the Running state, the system expects further messages. If a shutdown() message is received, the object returns to the shutdown state.

3. If a reportWeather() message is received, the system moves to the Summarizing state. When the summary is complete, the system moves to a Transmitting state where the information is transmitted to the remote system. It then returns to the Running state.

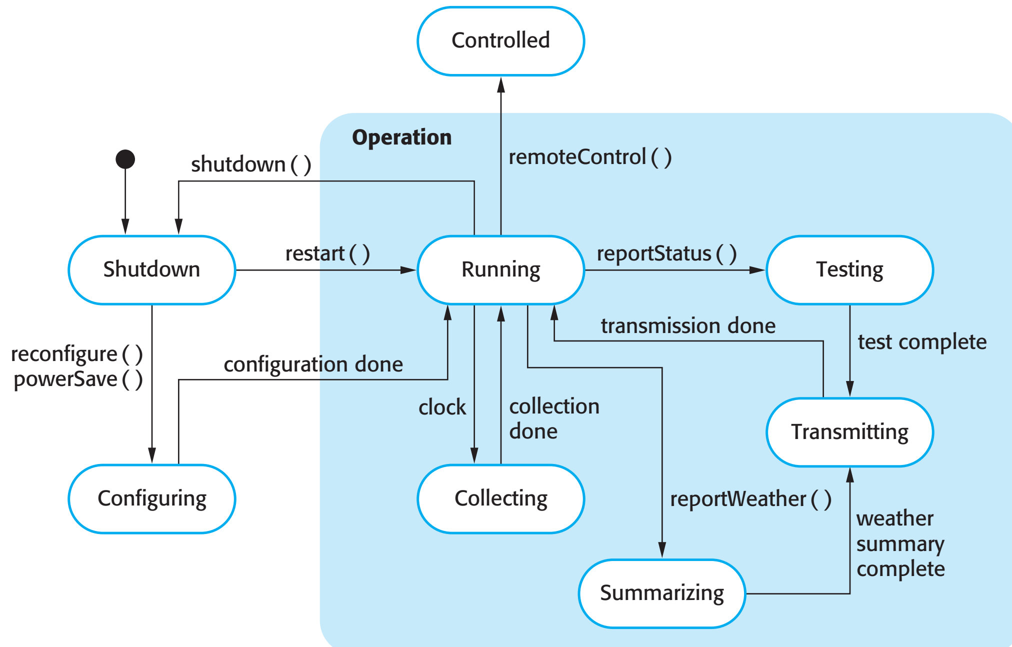



**Figure 7.8** Weather station state diagram

4. If a reportStatus() message is received, the system moves to the Testing state, then the Transmitting state, before returning to the Running state.

5. If a signal from the clock is received, the system moves to the Collecting state, where it collects data from the instruments. Each instrument is instructed in turn to collect its data from the associated sensors.

6. If a remoteControl() message is received, the system moves to a controlled state in which it responds to a different set of messages from the remote control room. These are not shown on this diagram.

State diagrams are useful high-level models of a system or an object's operation. You don't usually need a state diagram for all of the objects in the system. Many of the objects in a system are relatively simple and a state model adds unnecessary detail to the design.

### 7.1.5 Interface specification

An important part of any design process is the specification of the interfaces between the components in the design. You need to specify interfaces so that objects and sub-systems can be designed in parallel. Once an interface has been specified, the developers of other objects may assume that interface will be implemented.

Interface design is concerned with specifying the detail of the interface to an object or to a group of objects. This means defining the signatures and semantics of

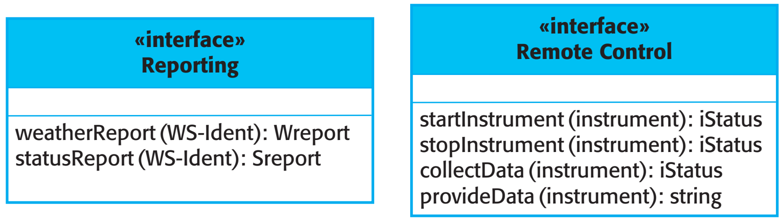



**Figure 7.9** Weather station interfaces

the services that are provided by the object or by a group of objects. Interfaces can be specified in the UML using the same notation as a class diagram. However, there is no attribute section and the UML stereotype «interface» should be included in the name part. The semantics of the interface may be defined using the object constraint language (OCL). I explain this in Chapter 17, where I cover component-based software engineering. I also show an alternative way to represent interfaces in the UML.

You should not include details of the data representation in an interface design, as attributes are not defined in an interface specification. However, you should include operations to access and update data. As the data representation is hidden, it can be easily changed without affecting the objects that use that data. This leads to a design that is inherently more maintainable. For example, an array representation of a stack may be changed to a list representation without affecting other objects that use the stack. By contrast, it often makes sense to expose the attributes in a static design model, as this is the most compact way of illustrating essential characteristics of the objects.

There is not a simple 1:1 relationship between objects and interfaces. The same object may have several interfaces, each of which is a viewpoint on the methods that it provides. This is supported directly in Java, where interfaces are declared separately from objects and objects 'implement' interfaces. Equally, a group of objects may all be accessed through a single interface.

Figure 7.9 shows two interfaces that may be defined for the weather station. The left-hand interface is a reporting interface that defines the operation names that are used to generate weather and status reports. These map directly to operations in the WeatherStation object. The remote control interface provides four operations, which map onto a single method in the WeatherStation object. In this case, the individual operations are encoded in the command string associated with the remoteControl method, shown in Figure 7.6.

## 7.2 Design patterns

Design patterns were derived from ideas put forward by Christopher Alexander (Alexander et al., 1977), who suggested that there were certain common patterns of building design 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

**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, you have to provide multiple displays of 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 should 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 is 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**: This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) 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 that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
The UML model of the pattern is shown in Figure 7.12.

**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, optimizations 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 7.10** The Observer pattern

different settings. The pattern is not a detailed specification. Rather, you can think of it as a description of accumulated wisdom and experience, a well-tried solution to a common problem.

A quote from the Hillside Group web site (http://hillside.net), which is dedicated to maintaining information about patterns, encapsulates their role in reuse:

> *Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience.*

Patterns have made a huge impact on object-oriented software design. As well as being tested solutions to common problems, they have become a vocabulary for talking about a design. You can therefore explain your design by describing the patterns that you have used. This is particularly true for the best-known design patterns that were originally described by the 'Gang of Four' in their patterns book, (Gamma et al., 1995). Other particularly important pattern descriptions are those published in a series of books by authors from Siemens, a large European technology company (Buschmann et al., 1996; Buschmann et al., 2007a; Buschmann et al., 2007b; Kircher and Jain, 2004; Schmidt et al., 2000).

Design patterns are usually associated with object-oriented design. Published patterns often rely on object characteristics such as inheritance and polymorphism to provide generality. However, the general principle of encapsulating experience in a

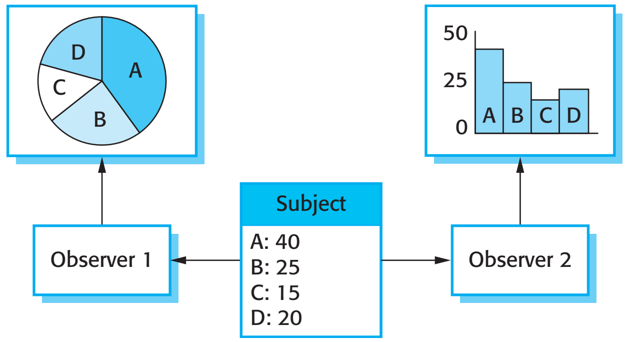



**Figure 7.11** Multiple displays

pattern is one that is equally applicable to any kind of software design. So, you could have configuration patterns for COTS systems. Patterns are a way of reusing the knowledge and experience of other designers.

The four essential elements of design patterns were defined by the 'Gang of Four' in their patterns book:

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 of the parts of the design solution, their relationships, and their 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 can help designers understand whether or not a pattern can be used in a particular situation.

Gamma and his co-authors break down the problem description into motivation (a description of why the pattern is useful) and applicability (a description of situations in which 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 use the Observer pattern, taken from the book by Gamma et al. (Gamma et al., 1995). This is shown in Figure 7.10. In my description, I use the four essential description elements and also include a brief statement of what the pattern can do. This pattern can be used in situations where different presentations of an object's state are required. It separates the object that must be displayed from the different forms of presentation. This is illustrated in Figure 7.11, which shows two graphical presentations of the same data set.

Graphical representations are normally used to illustrate the object classes in patterns and their relationships. These supplement the pattern description and add

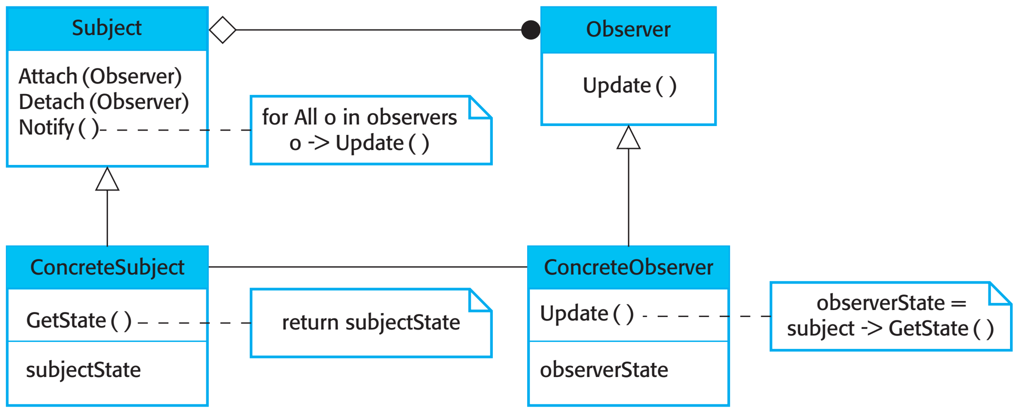



**Figure 7.12** A UML model of the Observer pattern

detail to the solution description. Figure 7.12 is the representation in UML of the Observer pattern.

To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied. Examples of such problems, documented in the 'Gang of Four's original patterns book, include:

1. Tell several objects that the state of some other object has changed (Observer pattern).

2. Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).

3. Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).

4. Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).

Patterns support high-level, concept reuse. When you try to reuse executable components you are inevitably constrained by detailed design decisions that have been made by the implementers 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. When these design decisions conflict with your particular requirements, reusing the component is either impossible or introduces inefficiencies into your system. Using patterns means that you reuse the ideas but can adapt the implementation to suit the system that you are developing.

When you start designing a system, it can be difficult to know, in advance, if you will need a particular pattern. Therefore, using patterns in a design process often involves developing a design, experiencing a problem, and then recognizing that a pattern can be used. This is certainly possible if you focus on the 23 general-purpose patterns documented in the original patterns book. However, if your problem is a different one, you may find it difficult to find an appropriate pattern amongst the hundreds of different patterns that have been proposed.

Patterns are a great idea but you need experience of software design to use them effectively. You have to recognize situations where a pattern can be applied. Inexperienced programmers, even if they have read the pattern books, will always find it hard to decide whether they can reuse a pattern or need to develop a special-purpose solution.

## 7.3 Implementation issues

Software engineering includes all of the activities involved in software development from the initial requirements of the system through to maintenance and management of the deployed system. A critical stage of this process is, of course, system implementation, where you create an executable version of the software. Implementation may involve developing programs in high- or low-level programming languages or tailoring and adapting generic, off-the-shelf systems to meet the specific requirements of an organization.

I assume that most readers of this book will understand programming principles and will have some programming experience. As this chapter is intended to offer a language-independent approach, I haven't focused on issues of good programming practice as this has to use language-specific examples. Instead, I introduce some aspects of implementation that are particularly important to software engineering that are often not covered in programming texts. These are:

1. *Reuse* Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.

2. *Configuration management* During the development process, many different versions of each software component are created. If you don't keep track of these versions in a configuration management system, you are liable to include the wrong versions of these components in your system.

3. *Host-target development* Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system). The host and target systems are sometimes of the same type but, often they are completely different.

### 7.3.1 Reuse

From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language. The only significant reuse or software was the reuse of functions and objects in programming language libraries. However, costs and schedule pressure meant that this approach became increasingly unviable, especially for commercial and Internet-based systems. Consequently, an approach to development based around the reuse of existing software emerged and is now generally used for business systems, scientific software, and, increasingly, in embedded systems engineering.

Software reuse is possible at a number of different levels:

1. *The abstraction level* At this level, you don't reuse software directly but rather use knowledge of successful abstractions in the design of your software. Design patterns and architectural patterns (covered in Chapter 6) are ways of representing abstract knowledge for reuse.

2. *The object level* At this level, you directly reuse objects from a library rather than writing the code yourself. To implement this type of reuse, you have to find appropriate libraries and discover if the objects and methods offer the functionality that you need. For example, if you need to process mail messages in a Java program, you may use objects and methods from a JavaMail library.

3. *The component level* Components are collections of objects and object classes that operate together to provide related functions and services. You often have to adapt and extend the component by adding some code of your own. An example of component-level reuse is where you build your user interface using a framework. This is a set of general object classes that implement event handling, display management, etc. You add connections to the data to be displayed and write code to define specific display details such as screen layout and colors.

4. *The system level* At this level, you reuse entire application systems. This usually involves some kind of configuration of these systems. This may be done by adding and modifying code (if you are reusing a software product line) or by using the system's own configuration interface. Most commercial systems are now built in this way where generic COTS (commercial off-the-shelf) systems are adapted and reused. Sometimes this approach may involve reusing several different systems and integrating these to create a new system.

By reusing existing software, you can develop new systems more quickly, with fewer development risks and also lower costs. As the reused software has been tested in other applications, it should be more reliable than new software. However, there are costs associated with reuse:

1. The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs. You may have to test the software to make sure that it will work in your environment, especially if this is different from its development environment.

2. Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.

3. The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.

4. The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed. Integrating reusable software from different providers can be difficult and expensive because the providers may make conflicting assumptions about how their respective software will be reused.

How to reuse existing knowledge and software should be the first thing you should think about when starting a software development project. You should consider the possibilities of reuse before designing the software in detail, as you may wish to adapt your design to reuse existing software assets. As I discussed in Chapter 2, in a reuse-oriented development process, you search for reusable elements then modify your requirements and design to make best use of these.

For a large number of application systems, software engineering really means software reuse. I therefore devote several chapters in the software technologies section of the book to this topic (Chapters 16, 17, and 19).

### 7.3.2 Configuration management

In software development, change happens all the time, so change management is absolutely essential. When a team of people are developing software, you have to make sure that team members don't interfere with each others' work. That is, if two people are working on a component, their changes have to be coordinated. Otherwise, one programmer may make changes and overwrite the other's work. You also have to ensure that everyone can access the most up-to-date versions of software components, otherwise developers may redo work that has already been done. When something goes wrong with a new version of a system, you have to be able to go back to a working version of the system or component.

Configuration management is the name given to the general process of managing a changing software system. The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system. There are, therefore, three fundamental configuration management activities:

1. Version management, where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers. They stop one developer overwriting code that has been submitted to the system by someone else.

2. System integration, where support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.

3. Problem tracking, where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

Software configuration management tools support each of the above activities. These tools may be designed to work together in a comprehensive change management system, such as ClearCase (Bellagio and Milligan, 2005). In integrated configuration management systems, version management, system integration, and problem-tracking tools are designed together. They share a user interface style and are integrated through a common code repository.

Alternatively, separate tools, installed in an integrated development environment, may be used. Version management may be supported using a version management system such as Subversion (Pilato et al., 2008), which can support multi-site, multi-team development. System integration support may be built into the language or rely on a separate toolset such as the GNU build system. This includes what is perhaps the best-known integration tool, Unix make. Bug tracking or issue tracking systems, such as Bugzilla, are used to report bugs and other issues and to keep track of whether or not these have been fixed.

Because of its importance in professional software engineering, I discuss change and configuration management in more detail in Chapter 25.

### 7.3.3 Host-target development

Most software development is based on a host-target model. Software is developed on one computer (the host), but runs on a separate machine (the target). More generally, we can talk about a development platform and an execution platform. A platform is more than just hardware. It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.

Sometimes, the development and execution platforms are the same, making it possible to develop the software and test it on the same machine. More commonly, however, they are different so that you need to either move your developed software to the execution platform for testing or run a simulator on your development machine.

Simulators are often used when developing embedded systems. You simulate hardware devices, such as sensors, and the events in the environment in which the system will be deployed. Simulators speed up the development process for embedded systems as each developer can have their own execution platform with no need to download the software to the target hardware. However, simulators are expensive to develop and so are only usually available for the most popular hardware architectures.

If the target system has installed middleware or other software that you need to use, then you need to be able to test the system using that software. It may be impractical to install that software on your development machine, even if it is the same as the target platform, because of license restrictions. In those circumstances, you need to transfer your developed code to the execution platform to test the system.

**UML deployment diagrams**

UML deployment diagrams show how software components are physically deployed on processors; that is, the deployment diagram shows the hardware and software in the system and the middleware used to connect the different components in the system. Essentially, you can think of deployment diagrams as a way of defining and documenting the target environment.

**http://www.SoftwareEngineering-9.com/Web/Deployment/**

A software development platform should provide a range of tools to support software engineering processes. These may include:

1. An integrated compiler and syntax-directed editing system that allows you to create, edit, and compile code.
2. A language debugging system.
3. Graphical editing tools, such as tools to edit UML models.
4. Testing tools, such as JUnit (Massol, 2003) that can automatically run a set of tests on a new version of a program.
5. Project support tools that help you organize the code for different development projects.

As well as these standard tools, your development system may include more specialized tools such as static analyzers (discussed in Chapter 15). Normally, development environments for teams also include a shared server that runs a change and configuration management system and, perhaps, a system to support requirements management.

Software development tools are often grouped to create an integrated development environment (IDE). An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface. Generally, IDEs are created to support development in a specific programming language such as Java. The language IDE may be developed specially, or may be an instantiation of a general-purpose IDE, with specific language-support tools.

A general-purpose IDE is a framework for hosting software tools that provides data management facilities for the software being developed, and integration mechanisms, that allow tools to work together. The best-known general-purpose IDE is the Eclipse environment (Carlson, 2005). This environment is based on a plug-in architecture so that it can be specialized for different languages and application domains (Clayberg and Rubel, 2006). Therefore, you can install Eclipse and tailor it for your specific needs by adding plug-ins. For example, you may add a set of plug-ins to support networked systems development in Java or embedded systems engineering using C.

As part of the development process, you need to make decisions about how the developed software will be deployed on the target platform. This is straightforward for embedded systems, where the target is usually a single computer. However, for distributed systems, you need to decide on the specific platforms where the components will be deployed. Issues that you have to consider in making this decision are:

1. *The hardware and software requirements of a component* If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.

2. *The availability requirements of the system* High-availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.

3. *Component communications* If there is a high level of communications traffic between components, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other. This reduces communications latency, the delay between the time a message is sent by one component and received by another.

You can document your decisions on hardware and software deployment using UML deployment diagrams, which show how software components are distributed across hardware platforms.

If you are developing an embedded system, you may have to take into account target characteristics, such as its physical size, power capabilities, the need for real-time responses to sensor events, the physical characteristics of actuators, and its real-time operating system. I discuss embedded systems engineering in Chapter 20.

## 7.4 Open source development

Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process (Raymond, 2001). Its roots are in the Free Software Foundation (http://www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish. There was an assumption that the code would be controlled and developed by a small core group, rather than users of the code.

Open source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code. In principle at least, any contributor to an open source project may report and fix bugs and propose new features and functionality. However, in practice, successful open source systems still rely on a core group of developers who control changes to the software.

The best-known open source product is, of course, the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment. Other important open source products are Java, the Apache web server, and the mySQL database management system. Major players in the computer industry such as IBM and Sun support the open source movement and base their software on open source products. There are thousands of other, lesser known open source systems and components that may also be used.

It is usually fairly cheap or free to acquire open source software. You can normally download open source software without charge. However, if you want documentation and support, then you may have to pay for this, but costs are usually fairly low. The other key benefit of using open source products is that mature open source systems are usually very reliable. The reason for this is that they have a large population of users who are willing to fix problems themselves rather than report these problems to the developer and wait for a new release of the system. Bugs are discovered and repaired more quickly than is usually possible with proprietary software.

For a company involved in software development, there are two open source issues that have to be considered:

1. Should the product that is being developed make use of open source components?
2. Should an open source approach be used for the software's development?

The answers to these questions depend on the type of software that is being developed and the background and experience of the development team.

If you are developing a software product for sale, then time to market and reduced costs are critical. If you are developing in a domain in which there are high-quality open source systems available, you can save time and money by using these systems. However, if you are developing software to a specific set of organizational requirements, then using open source components may not be an option. You may have to integrate your software with existing systems that are incompatible with available open source systems. Even then, however, it could be quicker and cheaper to modify the open source system rather than redevelop the functionality that you need.

More and more product companies are using an open source approach to development. Their business model is not reliant on selling a software product but rather on selling support for that product. They believe that involving the open source community will allow software to be developed more cheaply, more quickly, and will create a community of users for the software. Again, however, this is really only applicable for general software products rather than specific organizational applications.

Many companies believe that adopting an open source approach will reveal confidential business knowledge to their competitors and so are reluctant to adopt this development model. However, if you are working in a small company and you open source your software, this may reassure customers that they will be able to support the software if your company goes out of business.

Publishing the source code of a system does not mean that people from the wider community will necessarily help with its development. Most successful open source products have been platform products rather than application systems. There are a limited number of developers who might be interested in specialized application systems. As such, making a software system open source does not guarantee community involvement.

### 7.4.1 Open source licensing

Although a fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code. Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it is used by including legally binding conditions in an open source software license (St. Laurent, 2004). Some open source developers believe that if an open source component is used to develop a new system, then that system should also be open source. Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed source systems.

Most open source licenses are derived from one of three general models:

1. The GNU General Public License (GPL). This is a so-called 'reciprocal' license that, simplistically, means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
2. The GNU Lesser General Public License (LGPL). This is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components. However, if you change the licensed component, then you must publish this as open source.
3. The Berkley Standard Distribution (BSD) License. This is a non-reciprocal license, which means you are not obliged to republish any changes or modifications made to open source code. You can include the code in proprietary systems that are sold. If you use open source components, you must acknowledge the original creator of the code.

Licensing issues are important because if you use open-source software as part of a software product, then you may be obliged by the terms of the license to make your own product open source. If you are trying to sell your software, you may wish to keep it secret. This means that you may wish to avoid using GPL-licensed open source software in its development.

If you are building software that runs on an open source platform, such as Linux, then licenses are not a problem. However, as soon as you start including open source components in your software you need to set up processes and databases to keep track of what's been used and their license conditions. Bayersdorfer (2007) suggests that companies managing projects that use open source should:

1. Establish a system for maintaining information about open source components that are downloaded and used. You have to keep a copy of the license for each

component that was valid at the time the component was used. Licenses may change so you need to know the conditions that you have agreed to.

2. Be aware of the different types of licenses and understand how a component is licensed before it is used. You may decide to use a component in one system but not in another because you plan to use these systems in different ways.

3. Be aware of evolution pathways for components. You need to know a bit about the open source project where components are developed to understand how they might change in future.

4. Educate people about open source. It's not enough to have procedures in place to ensure compliance with license conditions. You also need to educate developers about open source and open source licensing.

5. Have auditing systems in place. Developers, under tight deadlines, might be tempted to break the terms of a license. If possible, you should have software in place to detect and stop this.

6. Participate in the open source community. If you rely on open source products, you should participate in the community and help support their development.

The business model of software is changing. It is becoming increasingly difficult to build a business by selling specialized software systems. Many companies prefer to make their software open source and then sell support and consultancy to software users. This trend is likely to accelerate, with increasing use of open source software and with more and more software available in this form.

## KEY POINTS

- Software design and implementation are interleaved activities. The level of detail in the design depends on the type of system being developed and whether you are using a plan-driven or agile approach.
- The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models, and document the component interfaces.
- A range of different models may be produced during an object-oriented design process. These include static models (class models, generalization models, association models) and dynamic models (sequence models, state machine models).
- Component interfaces must be defined precisely so that other objects can use them. A UML interface stereotype may be used to define interfaces.
- When developing software, you should always consider the possibility of reusing existing software, either as components, services, or complete systems.

- Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating to develop software.
- Most software development is host-target development. You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.
- Open source development involves making the source code of a system publicly available. This means that many people can propose changes and improvements to the software.

## FURTHER READING

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

*Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development, 3rd edition*. Larman writes clearly on object-oriented design and, as well as discussing the use of the UML. This is a good introduction to using patterns in the design process. (C. Larman, Prentice Hall, 2004.)

*Producing Open Source Software: How to Run a Successful Free Software Project*. His book is a comprehensive guide to the background to open source software, licensing issues, and the practicalities of running an open source development project. (K. Fogel, O'Reilly Media Inc., 2008.)

Further reading on software reuse is suggested in Chapter 16 and on configuration management in Chapter 25.

## EXERCISES

**7.1.** Using the structured notation shown in Figure 7.3, specify the weather station use cases for Report status and Reconfigure. You should make reasonable assumptions about the functionality that is required here.

**7.2.** Assume that the MHC-PMS is being developed using an object-oriented approach. Draw a use case diagram showing at least six possible use cases for this system.

**7.3.** Using the UML graphical notation for object classes, design the following object classes, identifying attributes and operations. Use your own experience to decide on the attributes and operations that should be associated with these objects.

- a telephone
- a printer for a personal computer
- a personal stereo system
- a bank account
- a library catalog

**7.4.** Using the weather station objects identified in Figure 7.6 as a starting point, identify further objects that may be used in this system. Design an inheritance hierarchy for the objects that you have identified.

**7.5.** Develop the design of the weather station to show the interaction between the data collection subsystem and the instruments that collect weather data. Use sequence diagrams to show this interaction.

**7.6.** Identify possible objects in the following systems and develop an object-oriented design for them. You may make any reasonable assumptions about the systems when deriving the design.

- A group diary and time management system is intended to support the timetabling of meetings and appointments across a group of co-workers. When an appointment is to be made that involves a number of people, the system finds a common slot in each of their diaries and arranges the appointment for that time. If no common slots are available, it interacts with the user to rearrange his or her personal diary to make room for the appointment.
- A filling station (gas station) is to be set up for fully automated operation. Drivers swipe their credit card through a reader connected to the pump; the card is verified by communication with a credit company computer, and a fuel limit is established. The driver may then take the fuel required. When fuel delivery is complete and the pump hose is returned to its holster, the driver's credit card account is debited with the cost of the fuel taken. The credit card is returned after debiting. If the card is invalid, the pump returns it before fuel is dispensed.

**7.7.** Draw a sequence diagram showing the interactions of objects in a group diary system when a group of people are arranging a meeting.

**7.8.** Draw a UML state diagram showing the possible state changes in either the group diary or the filling station system.

**7.9.** Using examples, explain why configuration management is important when a team of people are developing a software product.

**7.10.** A small company has developed a specialized product that it configures specially for each customer. New customers usually have specific requirements to be incorporated into their system, and they pay for these to be developed. The company has an opportunity to bid for a new contract, which would more than double its customer base. The new customer also wishes to have some involvement in the configuration of the system. Explain why, in these circumstances, it might be a good idea for the company owning the software to make it open source.

## REFERENCES

Abbott, R. (1983). 'Program Design by Informal English Descriptions'. *Comm. ACM*, **26** (11), 882–94.

Alexander, C., Ishikawa, S. and Silverstein, M. (1977). *A Pattern Language: Towns, Building, Construction*. Oxford: Oxford University Press.

Bayersdorfer, M. (2007). 'Managing a Project with Open Source Components'. *ACM Interactions*, **14** (6), 33–4.

Beck, K. and Cunningham, W. (1989). 'A Laboratory for Teaching Object-Oriented Thinking'. *Proc. OOPSLA'89* (Conference on Object-oriented Programming, Systems, Languages and Applications), ACM Press. 1–6.

Bellagio, D. E. and Milligan, T. J. (2005). *Software Configuration Management Strategies and IBM Rational Clearcase: A Practical Introduction*. Boston: Pearson Education (IBM Press).

Buschmann, F., Henney, K. and Schmidt, D. C. (2007a). *Pattern-oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. New York: John Wiley & Sons.

Buschmann, F., Henney, K. and Schmidt, D. C. (2007b). *Pattern-oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. New York: John Wiley & Sons.

Buschmann, F., Meunier, R., Rohnert, H. and Sommerlad, P. (1996). *Pattern-oriented Software Architecture Volume 1: A System of Patterns*. New York: John Wiley & Sons.

Carlson, D. (2005). *Eclipse Distilled*. Boston: Addison-Wesley.

Clayberg, E. and Rubel, D. (2006). *Eclipse: Building Commercial-Quality Plug-Ins*. Boston: Addison Wesley.

Coad, P. and Yourdon, E. (1990). *Object-oriented Analysis*. Englewood Cliffs, NJ: Prentice Hall.

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

Harel, D. (1987). 'Statecharts: A Visual Formalism for Complex Systems'. *Sci. Comput. Programming*, **8** (3), 231–74.

Kircher, M. and Jain, P. (2004). *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. New York: John Wiley & Sons.

Massol, V. (2003). *JUnit in Action*. Greenwich, CT: Manning Publications.

Pilato, C., Collins-Sussman, B. and Fitzpatrick, B. (2008). *Version Control with Subversion*. Sebastopol, Calif.: O'Reilly Media Inc.

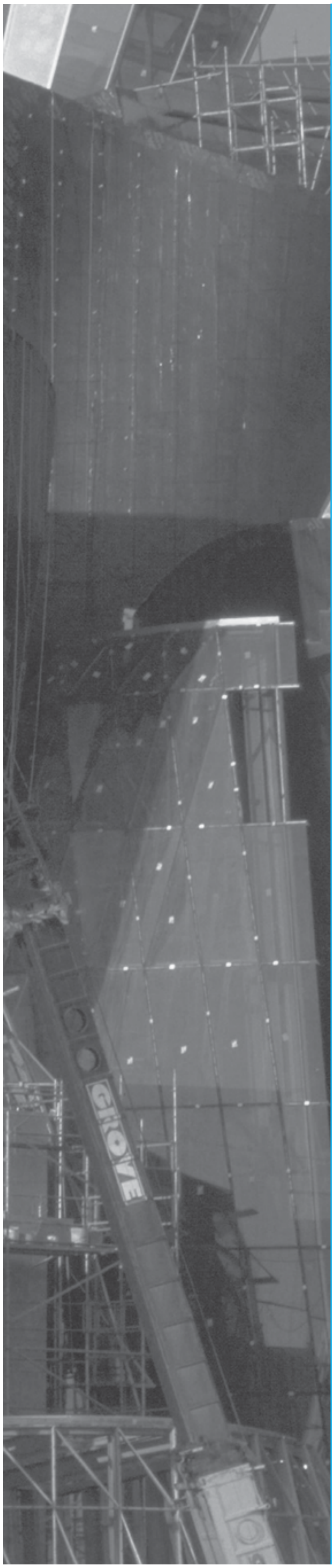
Raymond, E. S. (2001). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol. Calif.: O'Reilly Media, Inc.

Schmidt, D., Stal, M., Rohnert, H. and Buschmann, F. (2000). *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons.

Shlaer, S. and Mellor, S. (1988). *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, NJ: Yourdon Press.

St. Laurent, A. (2004). *Understanding Open Source and Free Software Licensing*. Sebastopol, Calif.: O'Reilly Media Inc.

Wirfs-Brock, R., Wilkerson, B. and Weiner, L. (1990). *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice Hall.



# 8

# Software testing

## Objectives

The objective of this chapter is to introduce software testing and software testing processes. When you have read the chapter, you will:

- understand the stages of testing from testing, during development to acceptance testing by system customers;
- have been introduced to techniques that help you choose test cases that are geared to discovering program defects;
- understand test-first development, where you design tests before writing code and run these tests automatically;
- know the important differences between component, system, and release testing and be aware of user testing processes and techniques.

## Contents

Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use. When you test software, you execute a program using artificial data. You check the results of the test run for errors, anomalies, or information about the program's non-functional attributes.

The testing process has two distinct goals:

1. To demonstrate to the developer and the customer that the software meets its requirements. For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.

2. To discover situations in which the behavior of the software is incorrect, undesirable, or does not conform to its specification. These are a consequence of software defects. Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations, and data corruption.

The first goal leads to validation testing, where you expect the system to perform correctly using a given set of test cases that reflect the system's expected use. The second goal leads to defect testing, where the test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used. Of course, there is no definite boundary between these two approaches to testing. During validation testing, you will find defects in the system; during defect testing, some of the tests will show that the program meets its requirements.

The diagram shown in Figure 8.1 may help to explain the differences between validation testing and defect testing. Think of the system being tested as a black box. The system accepts inputs from some input set I and generates outputs in an output set O. Some of the outputs will be erroneous. These are the outputs in set $O_e$ that are generated by the system in response to inputs in the set $I_e$. The priority in defect testing is to find those inputs in the set $I_e$ because these reveal problems with the system. Validation testing involves testing with correct inputs that are outside $I_e$. These stimulate the system to generate the expected correct outputs.

Testing cannot demonstrate that the software is free of defects or that it will behave as specified in every circumstance. It is always possible that a test that you have overlooked could discover further problems with the system. As Edsger Dijkstra, an early contributor to the development of software engineering, eloquently stated (Dijkstra et al., 1972):

*Testing can only show the presence of errors, not their absence*

Testing is part of a broader process of software verification and validation (V & V). Verification and validation are not the same thing, although they are often confused.

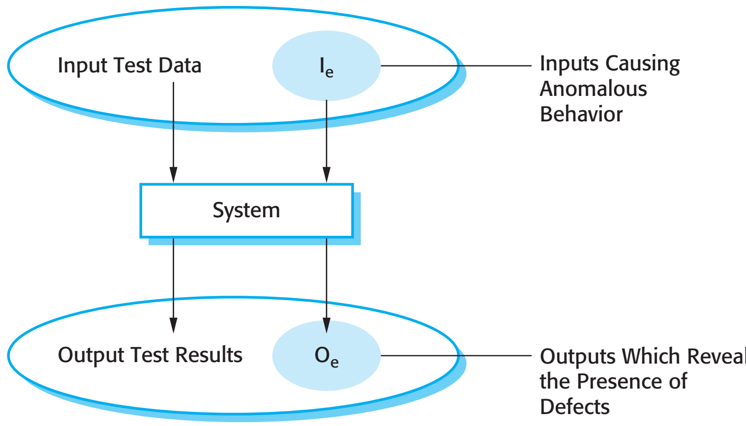



**Figure 8.1** An input-output model of program testing

Barry Boehm, a pioneer of software engineering, succinctly expressed the difference between them (Boehm, 1979):

- 'Validation: Are we building the right product?'
- 'Verification: Are we building the product right?'

Verification and validation processes are concerned with checking that software being developed meets its specification and delivers the functionality expected by the people paying for the software. These checking processes start as soon as requirements become available and continue through all stages of the development process.

The aim of verification is to check that the software meets its stated functional and non-functional requirements. Validation, however, is a more general process. The aim of validation is to ensure that the software meets the customer's expectations. It goes beyond simply checking conformance with the specification to demonstrating that the software does what the customer expects it to do. Validation is essential because, as I discussed in Chapter 4, requirements specifications do not always reflect the real wishes or needs of system customers and users.

The ultimate goal of verification and validation processes is to establish confidence that the software system is 'fit for purpose'. This means that the system must be good enough for its intended use. The level of required confidence depends on the system's purpose, the expectations of the system users, and the current marketing environment for the system:

1. *Software purpose* The more critical the software, the more important that it is reliable. For example, the level of confidence required for software used to control a safety-critical system is much higher than that required for a prototype that has been developed to demonstrate new product ideas.

2. *User expectations* Because of their experiences with buggy, unreliable software, many users have low expectations of software quality. They are not surprised when their software fails. When a new system is installed, users may tolerate