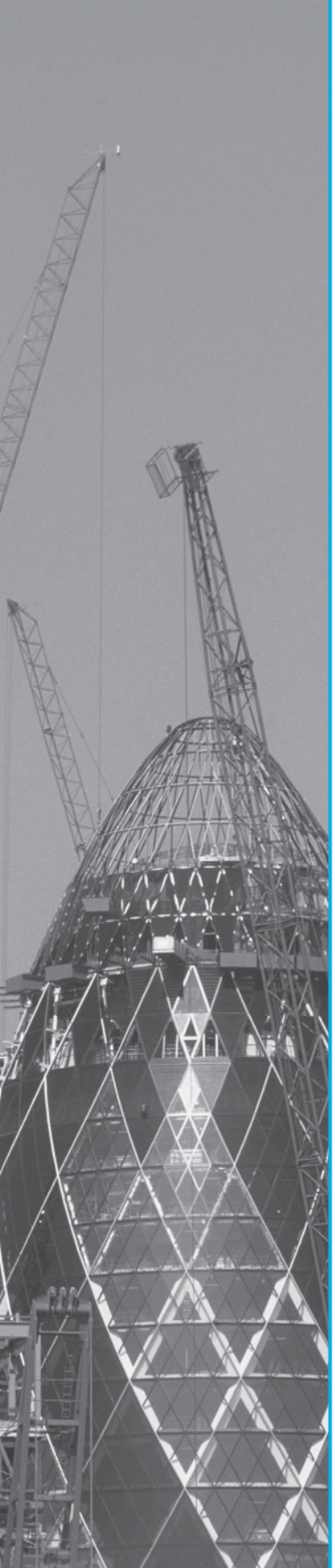


7 Design and implementation

Objectives

The objectives of this chapter are to introduce object-oriented software design using the UML and highlight important implementation concerns. When you have read this chapter, you will:

- understand the most important activities in a general, object-oriented design process;
- understand some of the different models that may be used to document an object-oriented design;
- know about the idea of design patterns and how these are a way of reusing design knowledge and experience;
- have been introduced to key issues that have to be considered when implementing software, including software reuse and open-source development.

Contents

Software design and implementation is the stage in the software engineering process at which an executable software system is developed. For some simple systems, software engineering means software design and implementation and all other software engineering activities are merged with this process. However, for large systems, software design and implementation is only one of a number of software engineering processes (requirements engineering, verification and validation, etc.).

Software design and implementation activities are invariably interleaved. Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements. Implementation is the process of realizing the design as a program. Sometimes there is a separate design stage, and this design is modeled and documented. At other times, a design is in the programmer's head or roughly sketched on a whiteboard or sheets of paper. Design is about how to solve a problem, so there is always a design process. However, it isn't always necessary or appropriate to describe the design in detail using the UML or other design description language.

Design and implementation are closely linked, and you should normally take implementation issues into account when developing a design. For example, using the UML to document a design may be the right thing to do if you are programming in an object-oriented language such as Java or C#. It is less useful, I think, if you are developing using a dynamically typed language like Python. There is no point in using the UML if you are implementing your system by configuring an off-the-shelf package. As I discussed in Chapter 3, agile methods usually work from informal sketches of the design and leave design decisions to programmers.

One of the most important implementation decisions that has to be made at an early stage of a software project is whether to build or to buy the application software. For many types of application, it is now possible to buy off-the-shelf application systems that can be adapted and tailored to the users' requirements. For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It is usually cheaper and faster to use this approach rather than developing a new system in a conventional programming language.

When you develop an application system by reusing an off-the-shelf product, the design process focuses on how to configure the system product to meet the application requirements. You don't develop design models of the system, such as models of the system objects and their interactions. I discuss this reuse-based approach to development in Chapter 15.

I assume that most readers of this book have had experience of program design and implementation. This is something that you acquire as you learn to program and master the elements of a programming language like Java or Python. You will have probably learned about good programming practice in the programming languages that you have studied, as well as how to debug programs that you have developed. Therefore, I don't cover programming topics here. Instead, this chapter has two aims:

1. To show how system modeling and architectural design (covered in Chapters 5 and 6) are put into practice in developing an object-oriented software design.

2. To introduce important implementation issues that are not usually covered in programming books. These include software reuse, configuration management and open-source development.

As there are a vast number of different development platforms, the chapter is not biased toward any particular programming language or implementation technology. Therefore, I have presented all examples using the UML rather than a programming language such as Java or Python.

7.1 Object-oriented design using the UML

An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state. The representation of the state is private and cannot be accessed directly from outside the object. Object-oriented design processes involve designing object classes and the relationships between these classes. These classes define the objects in the system and their interactions. When the design is realized as an executing program, the objects are created dynamically from these class definitions.

Objects include both data and operations to manipulate that data. They may therefore be understood and modified as stand-alone entities. Changing the implementation of an object or adding services should not affect other system objects. Because objects are associated with things, there is often a clear mapping between real-world entities (such as hardware components) and their controlling objects in the system. This improves the understandability, and hence the maintainability, of the design.

To develop a system design from concept to detailed, object-oriented design, you need to:

1. Understand and define the context and the external interactions with the system.
2. Design the system architecture.
3. Identify the principal objects in the system.
4. Develop design models.
5. Specify interfaces.

Like all creative activities, design is not a clear-cut, sequential process. You develop a design by getting ideas, proposing solutions, and refining these solutions as information becomes available. You inevitably have to backtrack and retry when problems arise. Sometimes you explore options in detail to see if they work; at other times you ignore details until late in the process. Sometimes you use notations, such as the UML, precisely to clarify aspects of the design; at other times, notations are used informally to stimulate discussions.

I explain object-oriented software design by developing a design for part of the embedded software for the wilderness weather station that I introduced in Chapter 1. Wilderness weather stations are deployed in remote areas. Each weather station

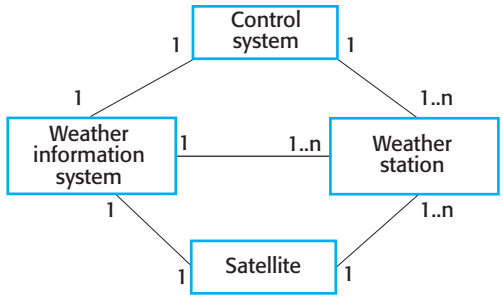


Figure 7.1 System context for the weather station

records local weather information and periodically transfers this to a weather information system, using a satellite link.

7.1.1 System context and interactions

The first stage in any software design process is to develop an understanding of the relationships between the software that is being designed and its external environment. This is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment. As I discussed in Chapter 5, understanding the context also lets you establish the boundaries of the system.

Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems. In this case, you need to decide how functionality is distributed between the control system for all of the weather stations and the embedded software in the weather station itself.

System context models and interaction models present complementary views of the relationships between a system and its environment:

1. A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
2. An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

The context model of a system may be represented using associations. Associations simply show that there are some relationships between the entities involved in the association. You can document the environment of the system using a simple block diagram, showing the entities in the system and their associations. Figure 7.1 shows that the systems in the environment of each weather station are a weather information system, an onboard satellite system, and a control system. The cardinality information on the link shows that there is a single control system but several weather stations, one satellite, and one general weather information system.

When you model the interactions of a system with its environment, you should use an abstract approach that does not include too much detail. One way to do this is to use a use case model. As I discussed in Chapters 4 and 5, each use case represents

Weather station use cases

Report weather–send weather data to the weather information system

Report status–send status information to the weather information system

Restart–if the weather station is shut down, restart the system

Shutdown–shut down the weather station

Reconfigure–reconfigure the weather station software

Powersave–put the weather station into power-saving mode

Remote control–send control commands to any weather station subsystem

http://software-engineering-book.com/web/ws-use-cases/

an interaction with the system. Each possible interaction is named in an ellipse, and the external entity involved in the interaction is represented by a stick figure.

The use case model for the weather station is shown in Figure 7.2. This shows that the weather station interacts with the weather information system to report weather data and the status of the weather station hardware. Other interactions are with a control system that can issue specific weather station control commands. The stick figure is used in the UML to represent other systems as well as human users.

Each of these use cases should be described in structured natural language. This helps designers identify objects in the system and gives them an understanding of what the system is intended to do. I use a standard format for this description that clearly identifies what information is exchanged, how the interaction is initiated, and so on. As I explain in Chapter 21, embedded systems are often modeled by describing

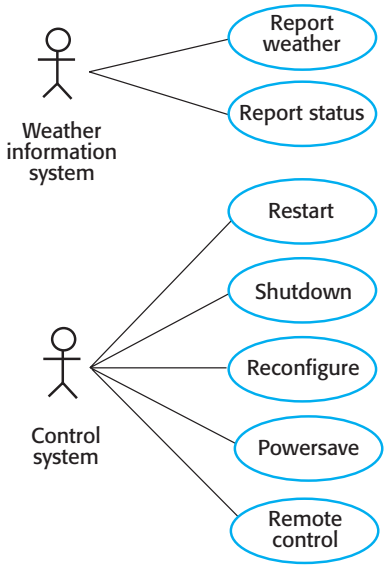


Figure 7.2 Weather station use cases

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Data	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum and average wind speeds; the total rainfall; and the wind direction as sampled at 5-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour, but this frequency may differ from one station to another and may be modified in future.

Figure 7.3 Use case description—Report weather

how they respond to internal or external stimuli. Therefore, the stimuli and associated responses should be listed in the description. Figure 7.3 shows the description of the Report weather use case from Figure 7.2 that is based on this approach.

7.1.2 Architectural design

Once the interactions between the software system and the system's environment have been defined, you use this information as a basis for designing the system architecture. Of course, you need to combine this knowledge with your general knowledge of the principles of architectural design and with more detailed domain knowledge. You identify the major components that make up the system and their interactions. You may then design the system organization using an architectural pattern such as a layered or client–server model.

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

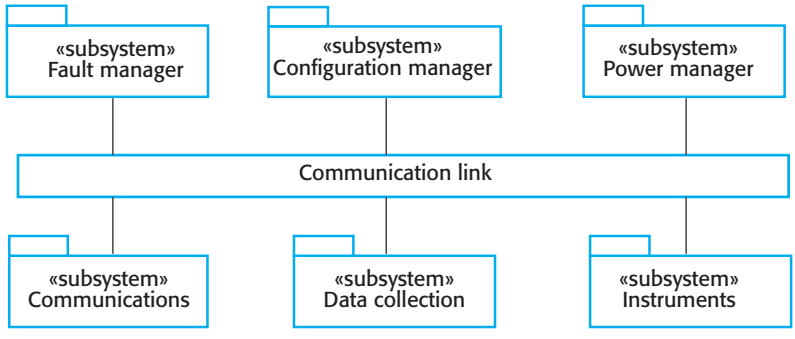


Figure 7.4 High-level architecture of weather station

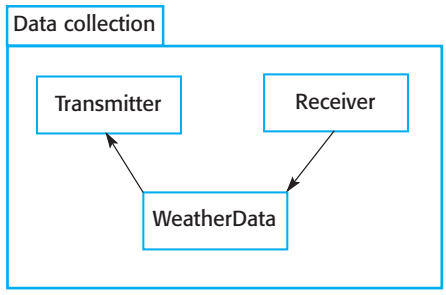


Figure 7.5 Architecture of data collection system

by broadcasting messages on a common infrastructure, shown as 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 "listener model" is a commonly used architectural style for distributed systems.

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 21.

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 you will need to implement objects representing the instruments that collect weather data and an object representing the summary of the weather data. You also usually need a high-level 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 general object classes in the system.

As object-oriented design evolved in the 1980s, various ways of identifying object classes in object-oriented systems were suggested:

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, events such as request, interactions such as meetings, locations

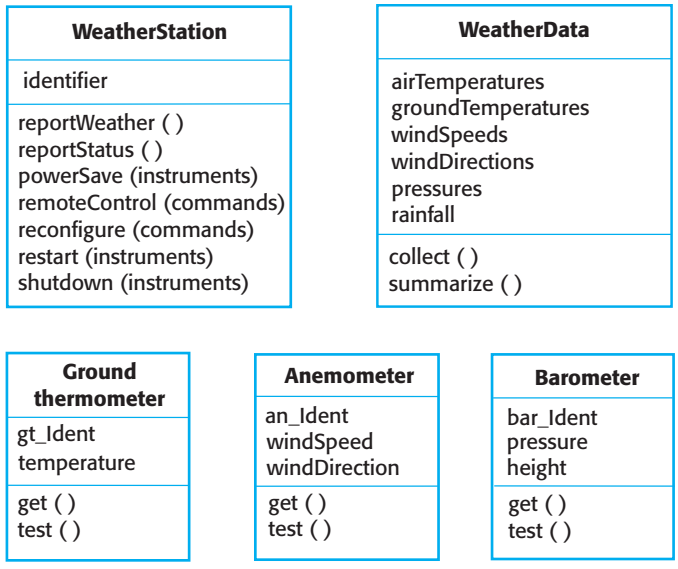


Figure 7.6 Weather station objects

such as offices, organizational units such as companies, and so on (Wirfs-Brock, Wilkerson, and Weiner 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. 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 analyses of existing systems. As well as the objects representing entities external to the system, you may also have to design "implementation objects" that are used to provide general services such as searching and validity checking.

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 are based on the interactions shown in Figure 7.3. I use a single object class, and it includes all of these interactions. Alternatively, you could design the system interface as several different classes, with one class per interaction.

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:

1. 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.
2. There are many remote weather stations, so each weather station should have its own identifier so that it can be uniquely identified in communications.
3. As weather stations are installed at different times, the types of instrument may be different. Therefore, each instrument should also be uniquely identified, and a database of instrument information should be maintained.

At this stage in the design process, you should focus on the objects themselves, without thinking about how these objects 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 records how often data should be collected.

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.

The level of detail that you need in a design model depends on the design process used. 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. Similarly, if agile development is used, outline design models on a whiteboard may be all that is required.

However, if a plan-based development process is used, you may need more detailed models. When the links between requirements engineers, designers, and programmers are indirect (e.g., where a system is being designed in one part of an organization but implemented elsewhere), then precise design descriptions are needed for communication. Detailed models, derived from the high-level abstract models, are used so that all team members have a common understanding of the design.

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. A sequential data-processing system is quite different from an embedded real-time system, so you need to use different types of design models. The UML supports 13 different types of models, but, as I discussed in Chapter 5, many of these models are not widely used. 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 should 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 expected runtime interactions between the system objects. Interactions that may be documented include the sequence of service requests made by objects and the state changes triggered by these object interactions.

I think three UML model types are particularly useful for adding detail to use case and architectural models:

1. *Subsystem models,* which 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 structural models.
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 models,* which show how 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 that 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 present the subsystems in the weather mapping system. As well as subsystem models, you may also design detailed object models, showing the objects in the systems and their associations (inheritance, generalization, aggregation, etc.). However, there is a danger

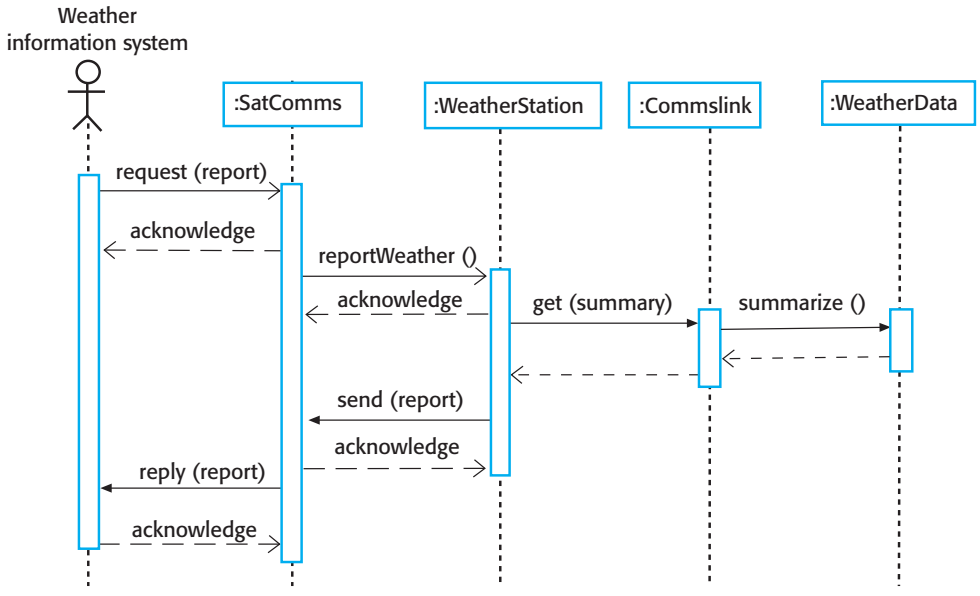


Figure 7.7 Sequence diagram describing data collection

in doing too much modeling. You should not make detailed decisions about the implementation that are really best left until the system is implemented.

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.

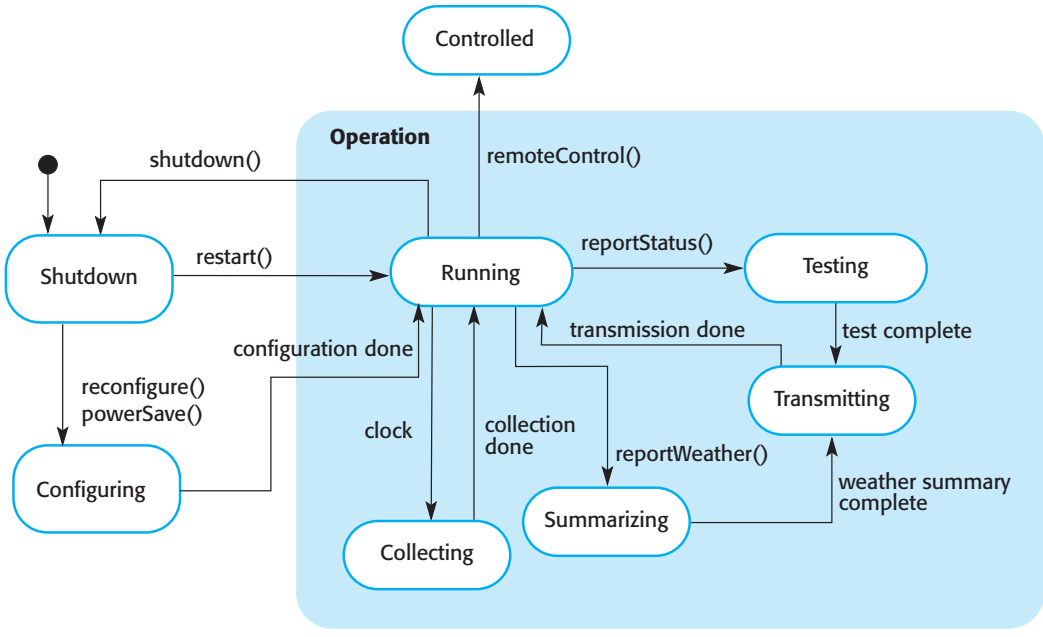


Figure 7.8 Weather station state diagram

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. As I discuss in Chapter 5, the UML includes state diagrams 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 allowed only 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.
4. 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.
5. 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. However, you don't need a state diagram for all of the objects in the system. Many system objects in a system are simple, and their operation can be easily described without a state model.

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 subsystems 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 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 discuss the use of the OCL in Chapter 16, where I explain how it can be used to describe the semantics of components.

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, you should normally expose the attributes in an object model, as this is the clearest way of describing the 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.

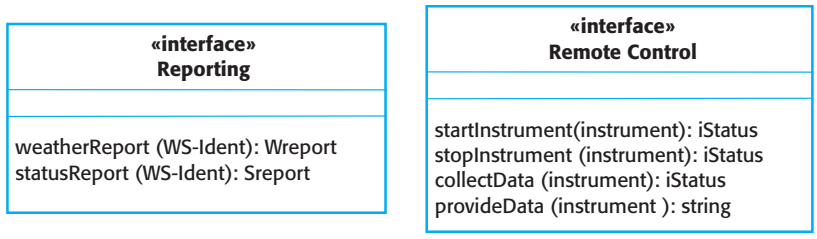


Figure 7.9 Weather station interfaces

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 1979), 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 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 website (hillside.net/patterns/), 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, published in 1995 (Gamma et al. 1995). Other important pattern descriptions are those published in a series of books by authors from Siemens, a large European technology company (Buschmann et al. 1996; Schmidt et al. 2000; Kircher and Jain 2004; Buschmann, Henney, and Schmidt 2007a, 2007b).

Patterns are a way of reusing the knowledge and experience of other designers. 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 pattern is

[†]The HIllside Group: hillside.net/patterns

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 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

one that is equally applicable to any kind of software design. For instance, you could have configuration patterns for instantiating reusable application systems.

The Gang of Four defined the four essential elements of design patterns in their book on patterns:

1. A name that is a meaningful reference to the pattern.
2. A description of the problem area that explains when the pattern may be applied.
3. A solution description 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 Gang of Four's patterns book. This is shown in Figure 7.10. In my description, I use the

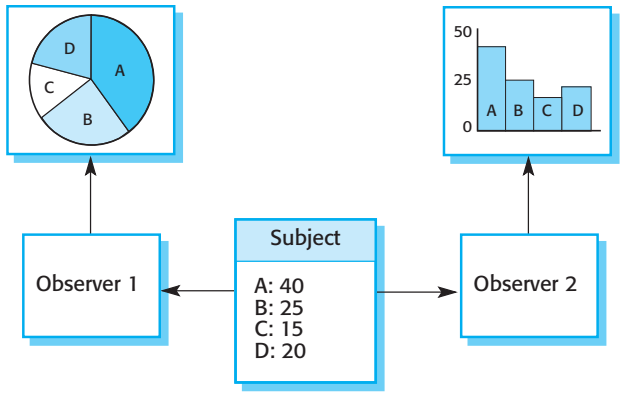


Figure 7.11 Multiple displays

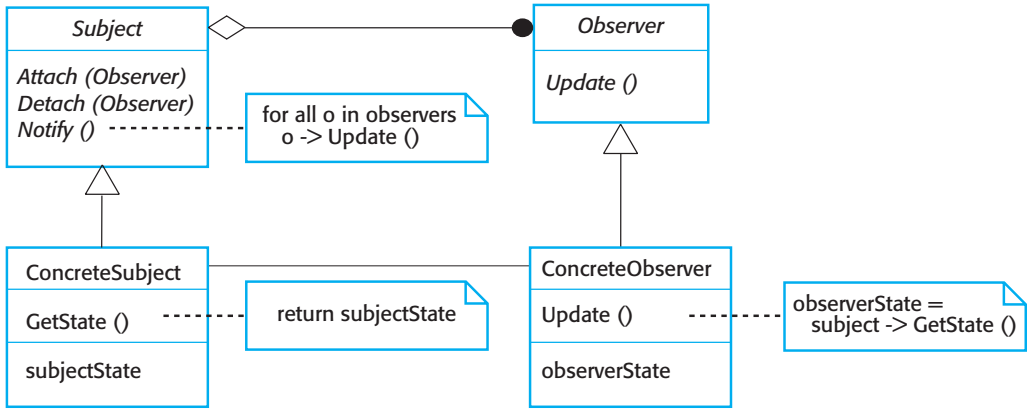
four essential description elements and also include a brief statement of what the pattern can do. This pattern can be used in situations where different presentations of an object's state are required. It separates the object that must be displayed from the different forms of presentation. This is illustrated in Figure 7.11, which shows two different graphical presentations of the same dataset.

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

Figure 7.12 A UML model of the Observer 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 runtime (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 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 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 among 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 language-specific examples need to be used. Instead, I introduce some aspects of implementation that are particularly important to software engineering and 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.

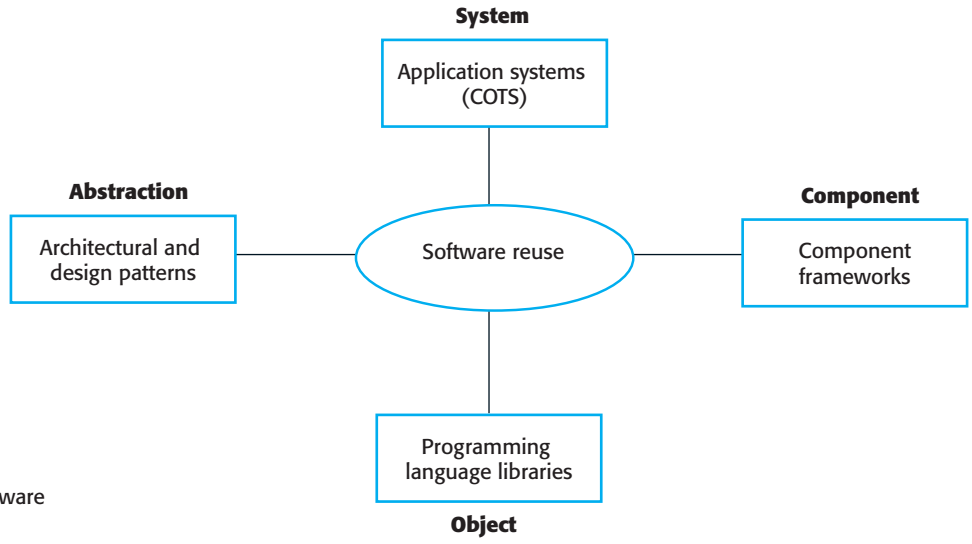


Figure 7.13 Software reuse

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 on the reuse of existing software is now the norm for many types of system development. A reuse-based approach is now widely used for web-based systems of all kinds, scientific software, and, increasingly, in embedded systems engineering.

Software reuse is possible at a number of different levels, as shown in Figure 7.13:

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 email 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 function 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 application systems systems are adapted and reused. Sometimes this approach may involve integrating several application systems to create a new system.

By reusing existing software, you can develop new systems more quickly, with fewer development risks and at lower cost. 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

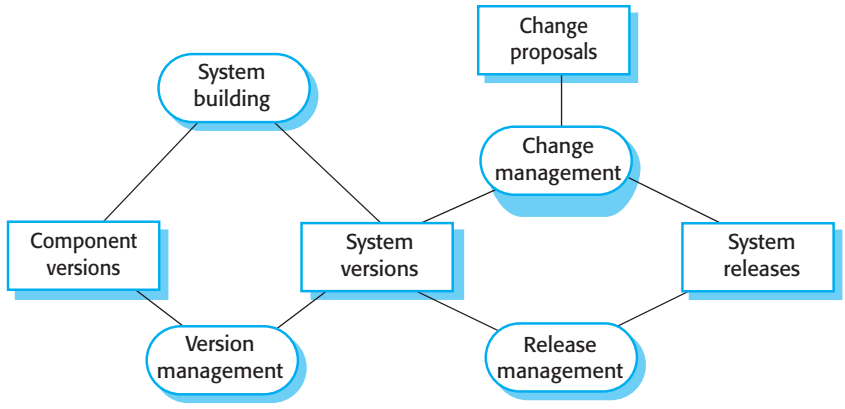


Figure 7.14 Configuration management

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 the best use of these.

Because of the importance of reuse in modern software engineering, I devote several chapters in Part 3 of this book to this topic (Chapters 15, 16, and 18).

7.3.2 Configuration management

In software development, change happens all the time, so change management is absolutely essential. When several people are involved in developing a software system, you have to make sure that team members don't interfere with each other's 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. As shown in Figure 7.14, there are four 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 from 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

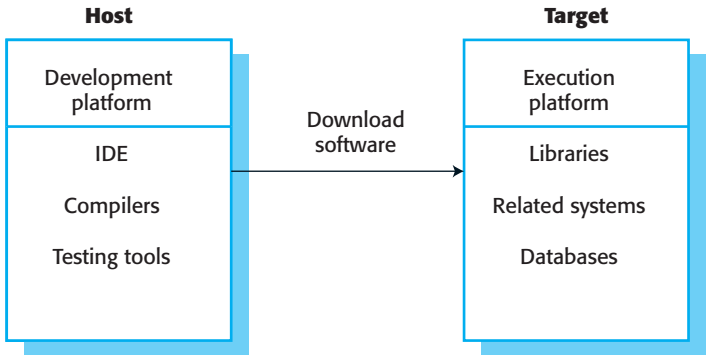


Figure 7.15 Host-target development

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.

4. *Release management,* where new versions of a software system are released to customers. Release management is concerned with planning the functionality of new releases and organizing the software for distribution.

Software configuration management tools support each of the above activities. These tools are usually installed in an integrated development environment, such as Eclipse. Version management may be supported using a version management system such as Subversion (Pilato, Collins-Sussman, and Fitzpatrick 2008) or Git (Loeliger and McCullough 2012), 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. 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. A comprehensive set of tools built around the Git system is available at Github (http://github.com).

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 professional software development is based on a host-target model (Figure 7.15). 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 (host) and an execution platform (target). 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 platform and execution platform are the same, making it possible to develop the software and test it on the same machine. Therefore, if you develop in Java, the target environment is the Java Virtual Machine. In principle, this is the same on every computer, so programs should be portable from one machine to another. However, particularly for embedded systems and mobile systems, the development and the execution platforms are different. 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 his or her own execution platform with no need to download the software to the target hardware. However, simulators are expensive to develop and so are usually available only 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. If this is the case, you need to transfer your developed code to the execution platform to test the system.

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, that can automatically run a set of tests on a new version of a program.
5. Tools to support refactoring and program visualization.
6. Configuration management tools to manage source code versions and to integrate and build systems.

In addition to these standard tools, your development system may include more specialized tools such as static analyzers (discussed in Chapter 12). 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 now usually installed within 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

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://software-engineering-book.com/web/deployment/

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 (http://www.eclipse.org). This environment is based on a plug-in architecture so that it can be specialized for different languages, such as Java, and application domains. 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 (Vogel 2013) 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 lot of intercomponent communication, it is usually best to deploy them on the same platform or on platforms that are physically close to one another. This reduces communications latency—the delay between the time that 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 21.

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 (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.

Open-source software is the backbone of the Internet and software engineering. The Linux operating system is the most widely used server system, as is the open-source Apache web server. Other important and universally used open-source products are Java, the Eclipse IDE, and the mySQL database management system. The Android operating system is installed on millions of mobile devices. Major players in the computer industry such as IBM and Oracle, support the open-source movement and base their software on open-source products. Thousands of other, lesser-known open-source systems and components may also be used.

It is usually cheap or even 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 widely used open-source systems are very reliable. 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 its own software 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 software 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.

Many software product companies are now using an open-source approach to development, especially for specialized systems. 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 and more quickly and will create a community of users for the software.

Some 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. Making a software system open source does not guarantee community involvement. There are thousands of open-source projects on Sourceforge and GitHub that have only a handful of downloads. However, if users of your software have concerns about its availability in future, making the software open source means that they can take their own copy and so be reassured that they will not lose access to it.

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) 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 (Chapman 2010) are variants of 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 nonreciprocal license, which means you are not obliged to re-publish 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. The MIT license is a variant of the BSD license with similar conditions.

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 but that does not reuse open-source components, then licenses are not a problem. However, if you embed open-source software in your software, you need processes and databases to keep track of what's been used and their license conditions. Bayersdorfer (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 open-source approach is one of several business models for software. In this model, companies release the source of their software and sell add-on services and advice in association with this. They may also sell cloud-based software services—an attractive option for users who do not have the expertise to manage their own open-source system and also specialized versions of their system for particular clients. Open-source is therefore likely to increase in importance as a way of developing and distributing software.