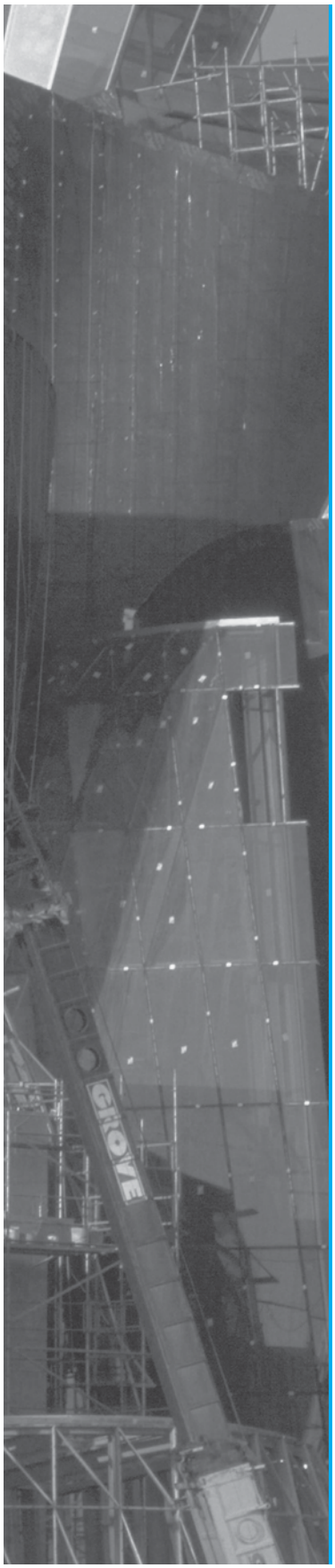


6 Architectural design

Objectives

The objective of this chapter is to introduce the concepts of software architecture and architectural design. When you have read the chapter, you will:

- understand why the architectural design of software is important;
- understand the decisions that have to be made about the system architecture during the architectural design process;
- have been introduced to the idea of architectural patterns, well-tried ways of organizing system architectures, which can be reused in system designs;
- know the architectural patterns that are often used in different types of application system, including transaction processing systems and language processing systems.

Contents

Architectural design is concerned with understanding how a system should be organized and designing the overall structure of that system. In the model of the software development process, as shown in Chapter 2, architectural design is the first stage in the software design process. It is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them. The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.

In agile processes, it is generally accepted that an early stage of the development process should be concerned with establishing an overall system architecture. Incremental development of architectures is not usually successful. While refactoring components in response to changes is usually relatively easy, refactoring a system architecture is likely to be expensive.

To help you understand what I mean by system architecture, consider Figure 6.1. This shows an abstract model of the architecture for a packing robot system that shows the components that have to be developed. This robotic system can pack different kinds of object. It uses a vision component to pick out objects on a conveyor, identify the type of object, and select the right kind of packaging. The system then moves objects from the delivery conveyor to be packaged. It places packaged objects on another conveyor. The architectural model shows these components and the links between them.

In practice, there is a significant overlap between the processes of requirements engineering and architectural design. Ideally, a system specification should not include any design information. This is unrealistic except for very small systems. Architectural decomposition is usually necessary to structure and organize the specification. Therefore, as part of the requirements engineering process, you might propose an abstract system architecture where you associate groups of system functions or features with large-scale components or sub-systems. You can then use this decomposition to discuss the requirements and features of the system with stakeholders.

You can design software architectures at two levels of abstraction, which I call *architecture in the small* and *architecture in the large*:

1. Architecture in the small is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components. This chapter is mostly concerned with program architectures.
2. Architecture in the large is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies. I cover architecture in the large in Chapters 18 and 19, where I discuss distributed systems architectures.

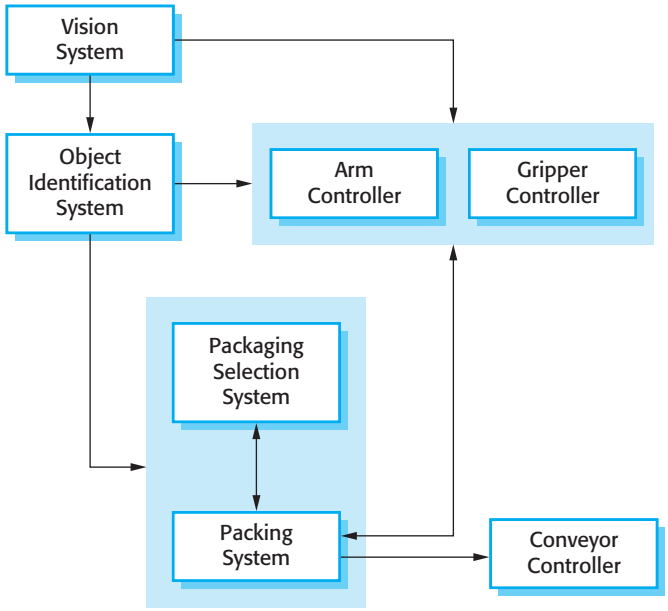


Figure 6.1 The architecture of a packing robot control system

Software architecture is important because it affects the performance, robustness, distributability, and maintainability of a system (Bosch, 2000). As Bosch discusses, individual components implement the functional system requirements. The non-functional requirements depend on the system architecture—the way in which these components are organized and communicate. In many systems, non-functional requirements are also influenced by individual components, but there is no doubt that the architecture of the system is the dominant influence.

Bass et al. (2003) discuss three advantages of explicitly designing and documenting software architecture:

1. *Stakeholder communication* The architecture is a high-level presentation of the system that may be used as a focus for discussion by a range of different stakeholders.
2. *System analysis* Making the system architecture explicit at an early stage in the system development requires some analysis. Architectural design decisions have a profound effect on whether or not the system can meet critical requirements such as performance, reliability, and maintainability.
3. *Large-scale reuse* A model of a system architecture is a compact, manageable description of how a system is organized and how the components interoperate. The system architecture is often the same for systems with similar requirements and so can support large-scale software reuse. As I explain in Chapter 16, it may be possible to develop product-line architectures where the same architecture is reused across a range of related systems.

Hofmeister et al. (2000) propose that a software architecture can serve firstly as a design plan for the negotiation of system requirements, and secondly as a means of structuring discussions with clients, developers, and managers. They also suggest that it is an essential tool for complexity management. It hides details and allows the designers to focus on the key system abstractions.

System architectures are often modeled using simple block diagrams, as in Figure 6.1. Each box in the diagram represents a component. Boxes within boxes indicate that the component has been decomposed to sub-components. Arrows mean that data and or control signals are passed from component to component in the direction of the arrows. You can see many examples of this type of architectural model in Booch's software architecture catalog (Booch, 2009).

Block diagrams present a high-level picture of the system structure, which people from different disciplines, who are involved in the system development process, can readily understand. However, in spite of their widespread use, Bass et al. (2003) dislike informal block diagrams for describing an architecture. They claim that these informal diagrams are poor architectural representations, as they show neither the type of the relationships among system components nor the components' externally visible properties.

The apparent contradictions between practice and architectural theory arise because there are two ways in which an architectural model of a program is used:

1. *As a way of facilitating discussion about the system design* A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail. The architectural model identifies the key components that are to be developed so managers can start assigning people to plan the development of these systems.
2. *As a way of documenting an architecture that has been designed* The aim here is to produce a complete system model that shows the different components in a system, their interfaces, and their connections. The argument for this is that such a detailed architectural description makes it easier to understand and evolve the system.

Block diagrams are an appropriate way of describing the system architecture during the design process, as they are a good way of supporting communications between the people involved in the process. In many projects, these are often the only architectural documentation that exists. However, if the architecture of a system is to be thoroughly documented then it is better to use a notation with well-defined semantics for architectural description. However, as I discuss in Section 6.2, some people think that detailed documentation is neither useful, nor really worth the cost of its development.

6.1 Architectural design decisions

Architectural design is a creative process where you design a system organization that will satisfy the functional and non-functional requirements of a system. Because it is a creative process, the activities within the process depend on the type of system being developed, the background and experience of the system architect, and the specific requirements for the system. It is therefore useful to think of architectural design as a series of decisions to be made rather than a sequence of activities.

During the architectural design process, system architects have to make a number of structural decisions that profoundly affect the system and its development process. Based on their knowledge and experience, they have to consider the following fundamental questions about the system:

1. Is there a generic application architecture that can act as a template for the system that is being designed?
2. How will the system be distributed across a number of cores or processors?
3. What architectural patterns or styles might be used?
4. What will be the fundamental approach used to structure the system?
5. How will the structural components in the system be decomposed into subcomponents?
6. What strategy will be used to control the operation of the components in the system?
7. What architectural organization is best for delivering the non-functional requirements of the system?
8. How will the architectural design be evaluated?
9. How should the architecture of the system be documented?

Although each software system is unique, systems in the same application domain often have similar architectures that reflect the fundamental concepts of the domain. For example, application product lines are applications that are built around a core architecture with variants that satisfy specific customer requirements. When designing a system architecture, you have to decide what your system and broader application classes have in common, and decide how much knowledge from these application architectures you can reuse. I discuss generic application architectures in Section 6.4 and application product lines in Chapter 16.

For embedded systems and systems designed for personal computers, there is usually only a single processor and you will not have to design a distributed architecture for the system. However, most large systems are now distributed systems in which the system software is distributed across many different computers. The choice of distribution architecture is a key decision that affects the performance and

reliability of the system. This is a major topic in its own right and I cover it separately in Chapter 18.

The architecture of a software system may be based on a particular architectural pattern or style. An architectural pattern is a description of a system organization (Garlan and Shaw, 1993), such as a client–server organization or a layered architecture. Architectural patterns capture the essence of an architecture that has been used in different software systems. You should be aware of common patterns, where they can be used, and their strengths and weaknesses when making decisions about the architecture of a system. I discuss a number of frequently used patterns in Section 6.3.

Garlan and Shaw's notion of an architectural style (style and pattern have come to mean the same thing) covers questions 4 to 6 in the previous list. You have to choose the most appropriate structure, such as client–server or layered structuring, that will enable you to meet the system requirements. To decompose structural system units, you decide on the strategy for decomposing components into sub-components. The approaches that you can use allow different types of architecture to be implemented. Finally, in the control modeling process, you make decisions about how the execution of components is controlled. You develop a general model of the control relationships between the various parts of the system.

Because of the close relationship between non-functional requirements and software architecture, the particular architectural style and structure that you choose for a system should depend on the non-functional system requirements:

1. *Performance* If performance is a critical requirement, the architecture should be designed to localize critical operations within a small number of components, with these components all deployed on the same computer rather than distributed across the network. This may mean using a few relatively large components rather than small, fine-grain components, which reduces the number of component communications. You may also consider run-time system organizations that allow the system to be replicated and executed on different processors.
2. *Security* If security is a critical requirement, a layered structure for the architecture should be used, with the most critical assets protected in the innermost layers, with a high level of security validation applied to these layers.
3. *Safety* If safety is a critical requirement, the architecture should be designed so that safety-related operations are all located in either a single component or in a small number of components. This reduces the costs and problems of safety validation and makes it possible to provide related protection systems that can safely shut down the system in the event of failure.
4. *Availability* If availability is a critical requirement, the architecture should be designed to include redundant components so that it is possible to replace and update components without stopping the system. I describe two fault-tolerant system architectures for high-availability systems in Chapter 13.
5. *Maintainability* If maintainability is a critical requirement, the system architecture should be designed using fine-grain, self-contained components that may

readily be changed. Producers of data should be separated from consumers and shared data structures should be avoided.

Obviously there is potential conflict between some of these architectures. For example, using large components improves performance and using small, fine-grain components improves maintainability. If both performance and maintainability are important system requirements, then some compromise must be found. This can sometimes be achieved by using different architectural patterns or styles for different parts of the system.

Evaluating an architectural design is difficult because the true test of an architecture is how well the system meets its functional and non-functional requirements when it is in use. However, you can do some evaluation by comparing your design against reference architectures or generic architectural patterns. Bosch's (2000) description of the non-functional characteristics of architectural patterns can also be used to help with architectural evaluation.

6.2 Architectural views

I explained in the introduction to this chapter that architectural models of a software system can be used to focus discussion about the software requirements or design. Alternatively, they may be used to document a design so that it can be used as a basis for more detailed design and implementation, and for the future evolution of the system. In this section, I discuss two issues that are relevant to both of these:

1. What views or perspectives are useful when designing and documenting a system's architecture?
2. What notations should be used for describing architectural models?

It is impossible to represent all relevant information about a system's architecture in a single architectural model, as each model only shows one view or perspective of the system. It might show how a system is decomposed into modules, how the run-time processes interact, or the different ways in which system components are distributed across a network. All of these are useful at different times so, for both design and documentation, you usually need to present multiple views of the software architecture.

There are different opinions as to what views are required. Krutchen (1995), in his well-known 4+1 view model of software architecture, suggests that there should be four fundamental architectural views, which are related using use cases or scenarios. The views that he suggests are:

1. A logical view, which shows the key abstractions in the system as objects or object classes. It should be possible to relate the system requirements to entities in this logical view.

2. A process view, which shows how, at run-time, the system is composed of interacting processes. This view is useful for making judgments about non-functional system characteristics such as performance and availability.
3. A development view, which shows how the software is decomposed for development, that is, it shows the breakdown of the software into components that are implemented by a single developer or development team. This view is useful for software managers and programmers.
4. A physical view, which shows the system hardware and how software components are distributed across the processors in the system. This view is useful for systems engineers planning a system deployment.

Hofmeister et al. (2000) suggest the use of similar views but add to this the notion of a conceptual view. This view is an abstract view of the system that can be the basis for decomposing high-level requirements into more detailed specifications, help engineers make decisions about components that can be reused, and represent a product line (discussed in Chapter 16) rather than a single system. Figure 6.1, which describes the architecture of a packing robot, is an example of a conceptual system view.

In practice, conceptual views are almost always developed during the design process and are used to support architectural decision making. They are a way of communicating the essence of a system to different stakeholders. During the design process, some of the other views may also be developed when different aspects of the system are discussed, but there is no need for a complete description from all perspectives. It may also be possible to associate architectural patterns, discussed in the next section, with the different views of a system.

There are differing views about whether or not software architects should use the UML for architectural description (Clements, et al., 2002). A survey in 2006 (Lange et al., 2006) showed that, when the UML was used, it was mostly applied in a loose and informal way. The authors of that paper argued that this was a bad thing. I disagree with this view. The UML was designed for describing object-oriented systems and, at the architectural design stage, you often want to describe systems at a higher level of abstraction. Object classes are too close to the implementation to be useful for architectural description.

I don't find the UML to be useful during the design process itself and prefer informal notations that are quicker to write and which can be easily drawn on a whiteboard. The UML is of most value when you are documenting an architecture in detail or using model-driven development, as discussed in Chapter 5.

A number of researchers have proposed the use of more specialized architectural description languages (ADLs) (Bass et al., 2003) to describe system architectures. The basic elements of ADLs are components and connectors, and they include rules and guidelines for well-formed architectures. However, because of their specialized nature, domain and application specialists find it hard to understand and use ADLs. This makes it difficult to assess their usefulness for practical software engineering. ADLs designed for a particular domain (e.g., automobile systems) may be used as a

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

Figure 6.2 The model-view-controller (MVC) pattern

basis for model-driven development. However, I believe that informal models and notations, such as the UML, will remain the most commonly used ways of documenting system architectures.

Users of agile methods claim that detailed design documentation is mostly unused. It is, therefore, a waste of time and money to develop it. I largely agree with this view and I think that, for most systems, it is not worth developing a detailed architectural description from these four perspectives. You should develop the views that are useful for communication and not worry about whether or not your architectural documentation is complete. However, an exception to this is when you are developing critical systems, when you need to make a detailed dependability analysis of the system. You may need to convince external regulators that your system conforms to their regulations and so complete architectural documentation may be required.

6.3 Architectural patterns

The idea of patterns as a way of presenting, sharing, and reusing knowledge about software systems is now widely used. The trigger for this was the publication of a book on object-oriented design patterns (Gamma et al., 1995), which prompted the development of other types of pattern, such as patterns for organizational design (Coplien and Harrison, 2004), usability patterns (Usability Group, 1998), interaction (Martin and Sommerville, 2004), configuration management (Berczuk and

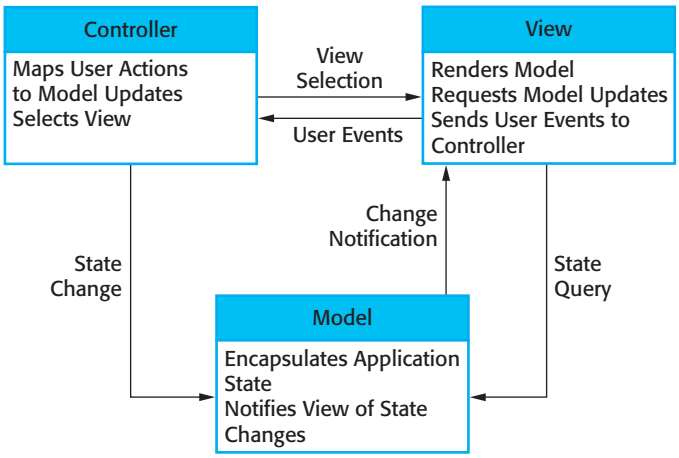


Figure 6.3 The organization of the MVC

Appleton, 2002), and so on. Architectural patterns were proposed in the 1990s under the name 'architectural styles' (Shaw and Garlan, 1996), with a five-volume series of handbooks on pattern-oriented software architecture published between 1996 and 2007 (Buschmann et al., 1996; Buschmann et al., 2007a; Buschmann et al., 2007b; Kircher and Jain, 2004; Schmidt et al., 2000).

In this section, I introduce architectural patterns and briefly describe a selection of architectural patterns that are commonly used in different types of systems. For more information about patterns and their use, you should refer to published pattern handbooks.

You can think of an architectural pattern as a stylized, abstract description of good practice, which has been tried and tested in different systems and environments. So, an architectural pattern should describe a system organization that has been successful in previous systems. It should include information of when it is and is not appropriate to use that pattern, and the pattern's strengths and weaknesses.

For example, Figure 6.2 describes the well-known Model-View-Controller pattern. This pattern is the basis of interaction management in many web-based systems. The stylized pattern description includes the pattern name, a brief description (with an associated graphical model), and an example of the type of system where the pattern is used (again, perhaps with a graphical model). You should also include information about when the pattern should be used and its advantages and disadvantages. Graphical models of the architecture associated with the MVC pattern are shown in Figures 6.3 and 6.4. These present the architecture from different views—Figure 6.3 is a conceptual view and Figure 6.4 shows a possible run-time architecture when this pattern is used for interaction management in a web-based system.

In a short section of a general chapter, it is impossible to describe all of the generic patterns that can be used in software development. Rather, I present some selected examples of patterns that are widely used and which capture good architectural design principles. I have included some further examples of generic architectural patterns on the book's web pages.

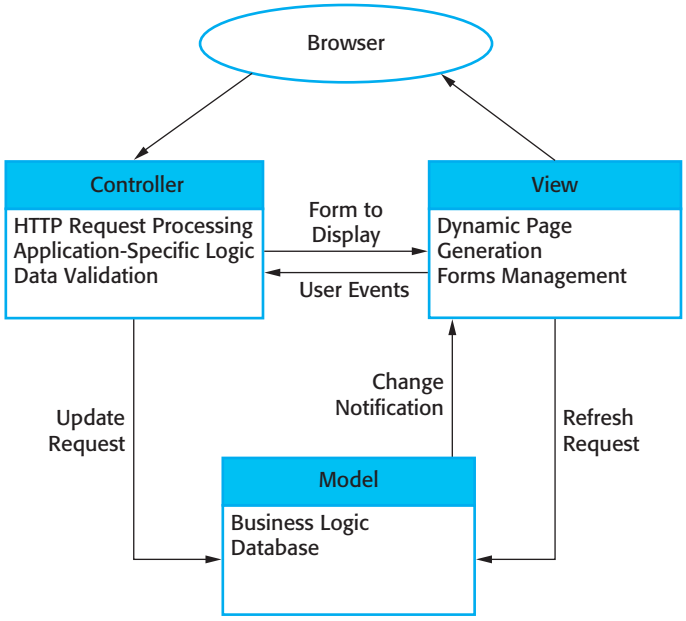


Figure 6.4 Web application architecture using the MVC pattern

6.3.1 Layered architecture

The notions of separation and independence are fundamental to architectural design because they allow changes to be localized. The MVC pattern, shown in Figure 6.2, separates elements of a system, allowing them to change independently. For example, adding a new view or changing an existing view can be done without any changes to the underlying data in the model. The layered architecture pattern is another way of achieving separation and independence. This pattern is shown in Figure 6.5. Here, the system functionality is organized into separate layers, and each layer only relies on the facilities and services offered by the layer immediately beneath it.

This layered approach supports the incremental development of systems. As a layer is developed, some of the services provided by that layer may be made available to users. The architecture is also changeable and portable. So long as its interface is unchanged, a layer can be replaced by another, equivalent layer. Furthermore, when layer interfaces change or new facilities are added to a layer, only the adjacent layer is affected. As layered systems localize machine dependencies in inner layers, this makes it easier to provide multi-platform implementations of an application system. Only the inner, machine-dependent layers need be re-implemented to take account of the facilities of a different operating system or database.

Figure 6.6 is an example of a layered architecture with four layers. The lowest layer includes system support software—typically database and operating system support. The next layer is the application layer that includes the components concerned with the application functionality and utility components that are used by other application components. The third layer is concerned with user interface

Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.
Example	A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

Figure 6.5 The layered architecture pattern

management and providing user authentication and authorization, with the top layer providing user interface facilities. Of course, the number of layers is arbitrary. Any of the layers in Figure 6.6 could be split into two or more layers.

Figure 6.7 is an example of how this layered architecture pattern can be applied to a library system called LIBSYS, which allows controlled electronic access to copyright material from a group of university libraries. This has a five-layer architecture, with the bottom layer being the individual databases in each library.

You can see another example of the layered architecture pattern in Figure 6.17 (found in Section 6.4). This shows the organization of the system for mental health-care (MHC-PMS) that I have discussed in earlier chapters.

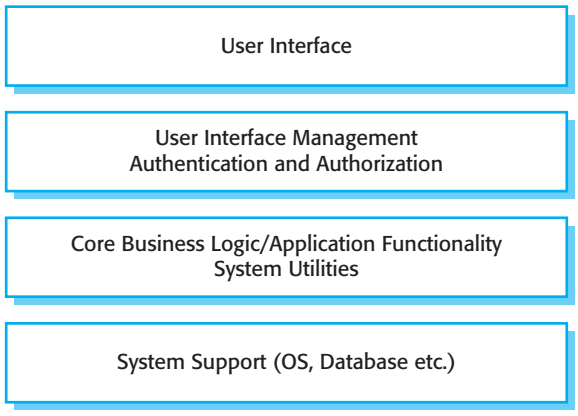


Figure 6.6 A generic layered architecture

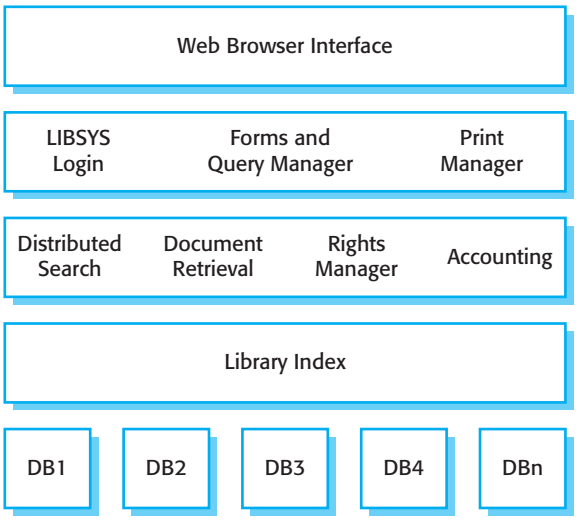


Figure 6.7 The architecture of the LIBSYS system

6.3.2 Repository architecture

The layered architecture and MVC patterns are examples of patterns where the view presented is the conceptual organization of a system. My next example, the Repository pattern (Figure 6.8), describes how a set of interacting components can share data.

The majority of systems that use large amounts of data are organized around a shared database or repository. This model is therefore suited to applications in which

Figure 6.8 The repository pattern

Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.9 is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

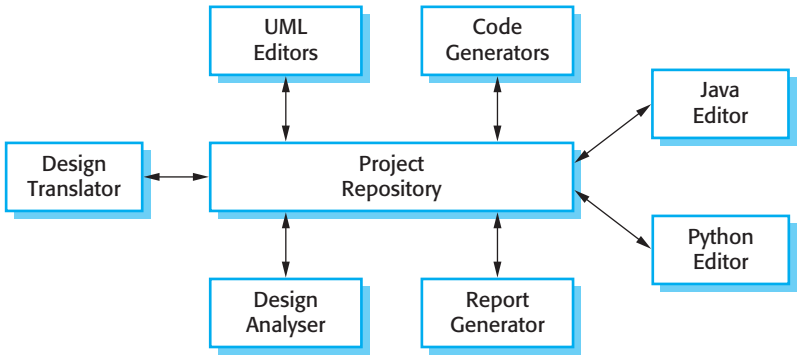


Figure 6.9 A repository architecture for an IDE

data is generated by one component and used by another. Examples of this type of system include command and control systems, management information systems, CAD systems, and interactive development environments for software.

Figure 6.9 is an illustration of a situation in which a repository might be used. This diagram shows an IDE that includes different tools to support model-driven development. The repository in this case might be a version-controlled environment (as discussed in Chapter 25) that keeps track of changes to software and allows roll-back to earlier versions.

Organizing tools around a repository is an efficient way to share large amounts of data. There is no need to transmit data explicitly from one component to another. However, components must operate around an agreed repository data model. Inevitably, this is a compromise between the specific needs of each tool and it may be difficult or impossible to integrate new components if their data models do not fit the agreed schema. In practice, it may be difficult to distribute the repository over a number of machines. Although it is possible to distribute a logically centralized repository, there may be problems with data redundancy and inconsistency.

In the example shown in Figure 6.9, the repository is passive and control is the responsibility of the components using the repository. An alternative approach, which has been derived for AI systems, uses a 'blackboard' model that triggers components when particular data become available. This is appropriate when the form of the repository data is less well structured. Decisions about which tool to activate can only be made when the data has been analyzed. This model is introduced by Nii (1986). Bosch (2000) includes a good discussion of how this style relates to system quality attributes.

6.3.3 Client–server architecture

The repository pattern is concerned with the static structure of a system and does not show its run-time organization. My next example illustrates a very commonly used run-time organization for distributed systems. The Client–server pattern is described in Figure 6.10.

Name	Client–server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

Figure 6.10 The client–server pattern

A system that follows the client–server pattern is organized as a set of services and associated servers, and clients that access and use the services. The major components of this model are:

1. A set of servers that offer services to other components. Examples of servers include print servers that offer printing services, file servers that offer file management services, and a compile server, which offers programming language compilation services.
2. A set of clients that call on the services offered by servers. There will normally be several instances of a client program executing concurrently on different computers.
3. A network that allows the clients to access these services. Most client–server systems are implemented as distributed systems, connected using Internet protocols.

Client–server architectures are usually thought of as distributed systems architectures but the logical model of independent services running on separate servers can be implemented on a single computer. Again, an important benefit is separation and independence. Services and servers can be changed without affecting other parts of the system.

Clients may have to know the names of the available servers and the services that they provide. However, servers do not need to know the identity of clients or how many clients are accessing their services. Clients access the services provided by a server through remote procedure calls using a request-reply protocol such as the http

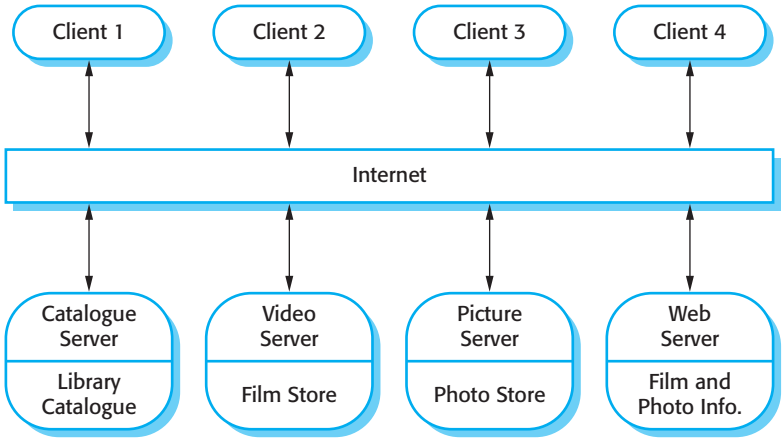


Figure 6.11 A client–server architecture for a film library

protocol used in the WWW. Essentially, a client makes a request to a server and waits until it receives a reply.

Figure 6.11 is an example of a system that is based on the client–server model. This is a multi-user, web-based system for providing a film and photograph library. In this system, several servers manage and display the different types of media. Video frames need to be transmitted quickly and in synchrony but at relatively low resolution. They may be compressed in a store, so the video server can handle video compression and decompression in different formats. Still pictures, however, must be maintained at a high resolution, so it is appropriate to maintain them on a separate server.

The catalog must be able to deal with a variety of queries and provide links into the web information system that includes data about the film and video clips, and an e-commerce system that supports the sale of photographs, film, and video clips. The

Figure 6.12 The pipe and filter pattern

Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.13 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

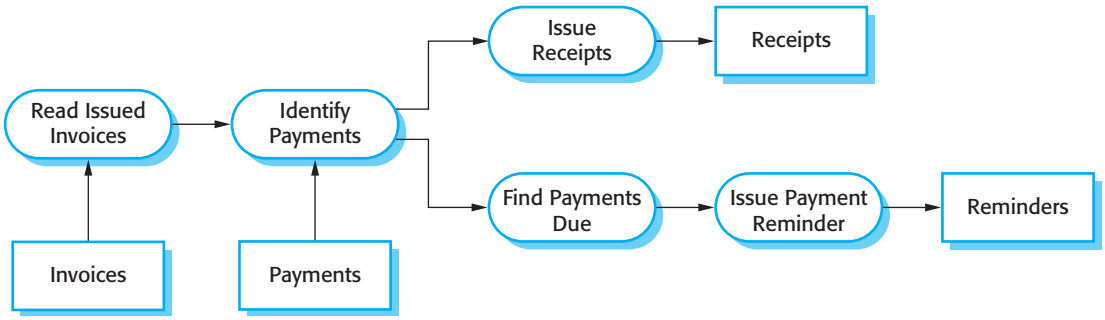


Figure 6.13 An example of the pipe and filter architecture

client program is simply an integrated user interface, constructed using a web browser, to access these services.

The most important advantage of the client–server model is that it is a distributed architecture. Effective use can be made of networked systems with many distributed processors. It is easy to add a new server and integrate it with the rest of the system or to upgrade servers transparently without affecting other parts of the system. I discuss distributed architectures, including client–server architectures and distributed object architectures, in Chapter 18.

6.3.4 Pipe and filter architecture

My final example of an architectural pattern is the pipe and filter pattern. This is a model of the run-time organization of a system where functional transformations process their inputs and produce outputs. Data flows from one to another and is transformed as it moves through the sequence. Each processing step is implemented as a transform. Input data flows through these transforms until converted to output. The transformations may execute sequentially or in parallel. The data can be processed by each transform item by item or in a single batch.

The name 'pipe and filter' comes from the original Unix system where it was possible to link processes using 'pipes'. These passed a text stream from one process to another. Systems that conform to this model can be implemented by combining Unix commands, using pipes and the control facilities of the Unix shell. The term 'filter' is used because a transformation 'filters out' the data it can process from its input data stream.

Variants of this pattern have been in use since computers were first used for automatic data processing. When transformations are sequential with data processed in batches, this pipe and filter architectural model becomes a batch sequential model, a common architecture for data processing systems (e.g., a billing system). The architecture of an embedded system may also be organized as a process pipeline, with each process executing concurrently. I discuss the use of this pattern in embedded systems in Chapter 20.

An example of this type of system architecture, used in a batch processing application, is shown in Figure 6.13. An organization has issued invoices to customers. Once a week, payments that have been made are reconciled with the invoices. For

Architectural patterns for control

There are specific architectural patterns that reflect commonly used ways of organizing control in a system. These include centralized control, based on one component calling other components, and event-based control, where the system reacts to external events.

http://www.SoftwareEngineering-9.com/Web/Architecture/ArchPatterns/

those invoices that have been paid, a receipt is issued. For those invoices that have not been paid within the allowed payment time, a reminder is issued.

Interactive systems are difficult to write using the pipe and filter model because of the need for a stream of data to be processed. Although simple textual input and output can be modeled in this way, graphical user interfaces have more complex I/O formats and a control strategy that is based on events such as mouse clicks or menu selections. It is difficult to translate this into a form compatible with the pipelining model.

6.4 Application architectures

Application systems are intended to meet a business or organizational need. All businesses have much in common—they need to hire people, issue invoices, keep accounts, and so on. Businesses operating in the same sector use common sector-specific applications. Therefore, as well as general business functions, all phone companies need systems to connect calls, manage their network, issue bills to customers, etc. Consequently, the application systems used by these businesses also have much in common.

These commonalities have led to the development of software architectures that describe the structure and organization of particular types of software systems. Application architectures encapsulate the principal characteristics of a class of systems. For example, in real-time systems, there might be generic architectural models of different system types, such as data collection systems or monitoring systems. Although instances of these systems differ in detail, the common architectural structure can be reused when developing new systems of the same type.

The application architecture may be re-implemented when developing new systems but, for many business systems, application reuse is possible without re-implementation. We see this in the growth of Enterprise Resource Planning (ERP) systems from companies such as SAP and Oracle, and vertical software packages (COTS) for specialized applications in different areas of business. In these systems, a generic system is configured and adapted to create a specific business application.

Application architectures

There are several examples of application architectures on the book's website. These include descriptions of batch data-processing systems, resource allocation systems, and event-based editing systems.

http://www.SoftwareEngineering-9.com/Web/Architecture/AppArch/

For example, a system for supply chain management can be adapted for different types of suppliers, goods, and contractual arrangements.

As a software designer, you can use models of application architectures in a number of ways:

1. *As a starting point for the architectural design process* If you are unfamiliar with the type of application that you are developing, you can base your initial design on a generic application architecture. Of course, this will have to be specialized for the specific system being developed, but it is a good starting point for design.
2. *As a design checklist* If you have developed an architectural design for an application system, you can compare this with the generic application architecture. You can check that your design is consistent with the generic architecture.
3. *As a way of organizing the work of the development team* The application architectures identify stable structural features of the system architectures and in many cases, it is possible to develop these in parallel. You can assign work to group members to implement different components within the architecture.
4. *As a means of assessing components for reuse* If you have components you might be able to reuse, you can compare these with the generic structures to see whether there are comparable components in the application architecture.
5. *As a vocabulary for talking about types of applications* If you are discussing a specific application or trying to compare applications of the same types, then you can use the concepts identified in the generic architecture to talk about the applications.

There are many types of application system and, in some cases, they may seem to be very different. However, many of these superficially dissimilar applications actually have much in common, and thus can be represented by a single abstract application architecture. I illustrate this here by describing the following architectures of two types of application:

1. *Transaction processing applications* Transaction processing applications are database-centered applications that process user requests for information and update the information in a database. These are the most common type of interactive business systems. They are organized in such a way that user actions can't interfere with each other and the integrity of the database is maintained. This

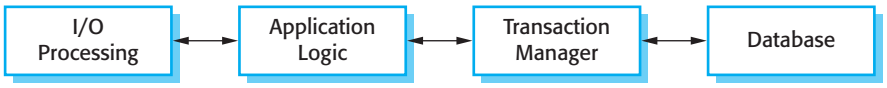


Figure 6.14 The structure of transaction processing applications

class of system includes interactive banking systems, e-commerce systems, information systems, and booking systems.

2. *Language processing systems* Language processing systems are systems in which the user's intentions are expressed in a formal language (such as Java). The language processing system processes this language into an internal format and then interprets this internal representation. The best-known language processing systems are compilers, which translate high-level language programs into machine code. However, language processing systems are also used to interpret command languages for databases and information systems, and markup languages such as XML (Harold and Means, 2002; Hunter et al., 2007).

I have chosen these particular types of system because a large number of web-based business systems are transaction-processing systems, and all software development relies on language processing systems.

6.4.1 Transaction processing systems

Transaction processing (TP) systems are designed to process user requests for information from a database, or requests to update a database (Lewis et al., 2003). Technically, a database transaction is sequence of operations that is treated as a single unit (an atomic unit). All of the operations in a transaction have to be completed before the database changes are made permanent. This ensures that failure of operations within the transaction does not lead to inconsistencies in the database.

From a user perspective, a transaction is any coherent sequence of operations that satisfies a goal, such as 'find the times of flights from London to Paris'. If the user transaction does not require the database to be changed then it may not be necessary to package this as a technical database transaction.

An example of a transaction is a customer request to withdraw money from a bank account using an ATM. This involves getting details of the customer's account, checking the balance, modifying the balance by the amount withdrawn, and sending commands to the ATM to deliver the cash. Until all of these steps have been completed, the transaction is incomplete and the customer accounts database is not changed.

Transaction processing systems are usually interactive systems in which users make asynchronous requests for service. Figure 6.14 illustrates the conceptual architectural structure of TP applications. First a user makes a request to the system through an I/O processing component. The request is processed by some application-specific logic. A transaction is created and passed to a transaction manager, which is usually embedded in the database management system. After the transaction manager

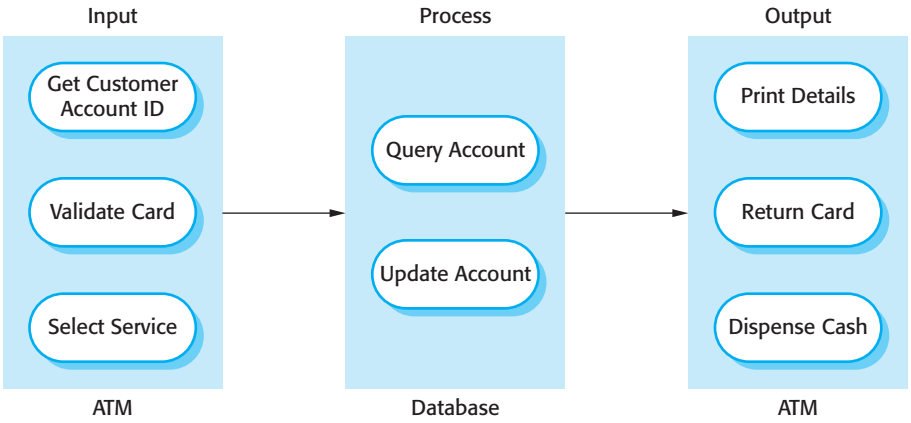


Figure 6.15 The software architecture of an ATM system

has ensured that the transaction is properly completed, it signals to the application that processing has finished.

Transaction processing systems may be organized as a 'pipe and filter' architecture with system components responsible for input, processing, and output. For example, consider a banking system that allows customers to query their accounts and withdraw cash from an ATM. The system is composed of two cooperating software components—the ATM software and the account processing software in the bank's database server. The input and output components are implemented as software in the ATM and the processing component is part of the bank's database server. Figure 6.15 shows the architecture of this system, illustrating the functions of the input, process, and output components.

6.4.2 Information systems

All systems that involve interaction with a shared database can be considered to be transaction-based information systems. An information system allows controlled access to a large base of information, such as a library catalog, a flight timetable, or the records of patients in a hospital. Increasingly, information systems are web-based systems that are accessed through a web browser.

Figure 6.16 a very general model of an information system. The system is modeled using a layered approach (discussed in Section 6.3) where the top layer supports the user interface and the bottom layer is the system database. The user communications layer handles all input and output from the user interface, and the information retrieval layer includes application-specific logic for accessing and updating the database. As we shall see later, the layers in this model can map directly onto servers in an Internet-based system.

As an example of an instantiation of this layered model, Figure 6.17 shows the architecture of the MHC-PMS. Recall that this system maintains and manages details of patients who are consulting specialist doctors about mental health problems. I have

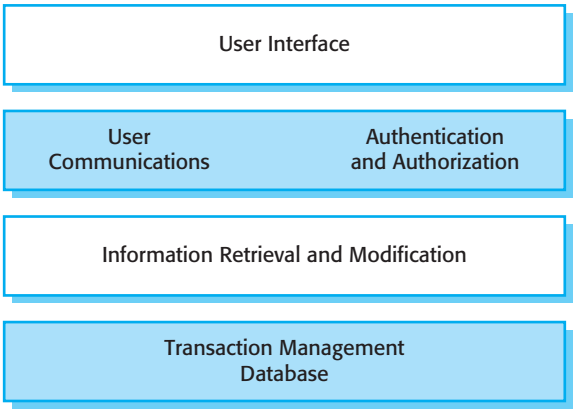


Figure 6.16 Layered information system architecture

added detail to each layer in the model by identifying the components that support user communications and information retrieval and access:

1. The top layer is responsible for implementing the user interface. In this case, the UI has been implemented using a web browser.
2. The second layer provides the user interface functionality that is delivered through the web browser. It includes components to allow users to log in to the system and checking components that ensure that the operations they use are allowed by their role. This layer includes form and menu management components that present information to users, and data validation components that check information consistency.
3. The third layer implements the functionality of the system and provides components that implement system security, patient information creation and updating, import and export of patient data from other databases, and report generators that create management reports.

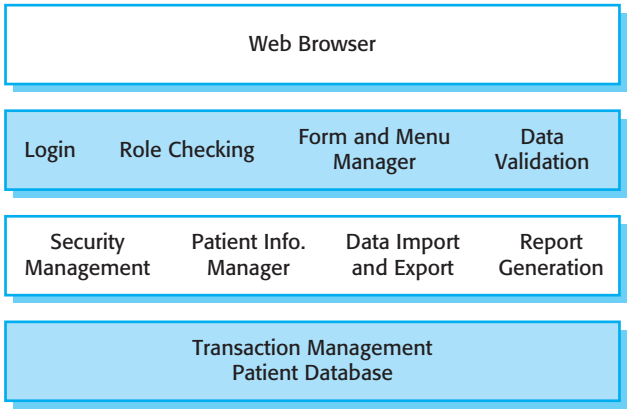


Figure 6.17 The architecture of the MHC-PMS

4. Finally, the lowest layer, which is built using a commercial database management system, provides transaction management and persistent data storage.

Information and resource management systems are now usually web-based systems where the user interfaces are implemented using a web browser. For example, e-commerce systems are Internet-based resource management systems that accept electronic orders for goods or services and then arrange delivery of these goods or services to the customer. In an e-commerce system, the application-specific layer includes additional functionality supporting a 'shopping cart' in which users can place a number of items in separate transactions, then pay for them all together in a single transaction.

The organization of servers in these systems usually reflects the four-layer generic model presented in Figure 6.16. These systems are often implemented as multi-tier client server/architectures, as discussed in Chapter 18:

1. The web server is responsible for all user communications, with the user interface implemented using a web browser;
2. The application server is responsible for implementing application-specific logic as well as information storage and retrieval requests;
3. The database server moves information to and from the database and handles transaction management.

Using multiple servers allows high throughput and makes it possible to handle hundreds of transactions per minute. As demand increases, servers can be added at each level to cope with the extra processing involved.

6.4.3 Language processing systems

Language processing systems translate a natural or artificial language into another representation of that language and, for programming languages, may also execute the resulting code. In software engineering, compilers translate an artificial programming language into machine code. Other language-processing systems may translate an XML data description into commands to query a database or to an alternative XML representation. Natural language processing systems may translate one natural language to another e.g., French to Norwegian.

A possible architecture for a language processing system for a programming language is illustrated in Figure 6.18. The source language instructions define the program to be executed and a translator converts these into instructions for an abstract machine. These instructions are then interpreted by another component that fetches the instructions for execution and executes them using (if necessary) data from the environment. The output of the process is the result of interpreting the instructions on the input data.

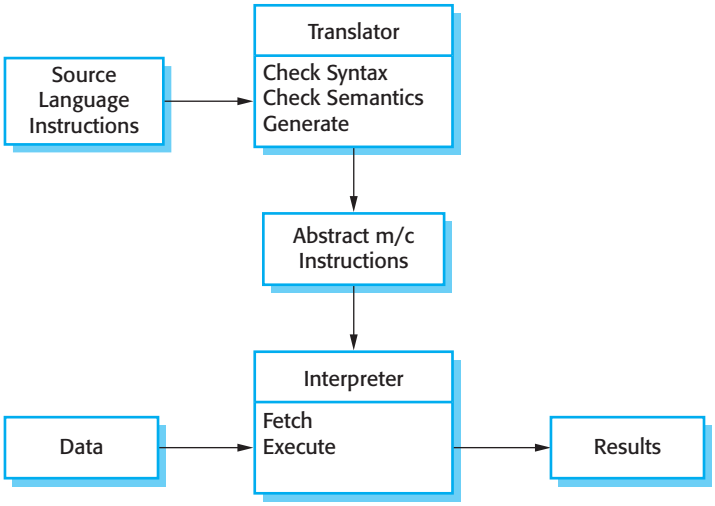


Figure 6.18 The architecture of a language processing system

Of course, for many compilers, the interpreter is a hardware unit that processes machine instructions and the abstract machine is a real processor. However, for dynamically typed languages, such as Python, the interpreter may be a software component.

Programming language compilers that are part of a more general programming environment have a generic architecture (Figure 6.19) that includes the following components:

1. A lexical analyzer, which takes input language tokens and converts them to an internal form.
2. A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
3. A syntax analyzer, which checks the syntax of the language being translated. It uses a defined grammar of the language and builds a syntax tree.
4. A syntax tree, which is an internal structure representing the program being compiled.

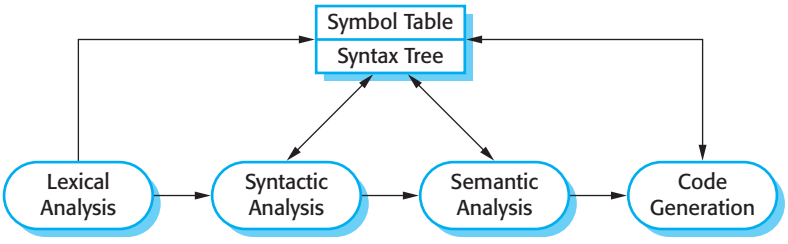


Figure 6.19 A pipe and filter compiler architecture

Reference architectures

Reference architectures capture important features of system architectures in a domain. Essentially, they include everything that might be in an application architecture although, in reality, it is very unlikely that any individual application would include all the features shown in a reference architecture. The main purpose of reference architectures is to evaluate and compare design proposals, and to educate people about architectural characteristics in that domain.

http://www.SoftwareEngineering-9.com/Web/Architecture/RefArch.html

5. A semantic analyzer that uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.
6. A code generator that 'walks' the syntax tree and generates abstract machine code.

Other components might also be included which analyze and transform the syntax tree to improve efficiency and remove redundancy from the generated machine code. In other types of language processing system, such as a natural language translator, there will be additional components such as a dictionary, and the generated code is actually the input text translated into another language.

There are alternative architectural patterns that may be used in a language processing system (Garlan and Shaw, 1993). Compilers can be implemented using a composite of a repository and a pipe and filter model. In a compiler architecture, the symbol table is a repository for shared data. The phases of lexical, syntactic, and semantic analysis are organized sequentially, as shown in Figure 6.19, and communicate through the shared symbol table.

This pipe and filter model of language compilation is effective in batch environments where programs are compiled and executed without user interaction; for example, in the translation of one XML document to another. It is less effective when a compiler is integrated with other language processing tools such as a structured editing system, an interactive debugger or a program prettyprinter. In this situation, changes from one component need to be reflected immediately in other components. It is better, therefore, to organize the system around a repository, as shown in Figure 6.20.

This figure illustrates how a language processing system can be part of an integrated set of programming support tools. In this example, the symbol table and syntax tree act as a central information repository. Tools or tool fragments communicate through it. Other information that is sometimes embedded in tools, such as the grammar definition and the definition of the output format for the program, have been taken out of the tools and put into the repository. Therefore, a syntax-directed editor can check that the syntax of a program is correct as it is being typed and a prettyprinter can create listings of the program in a format that is easy to read.

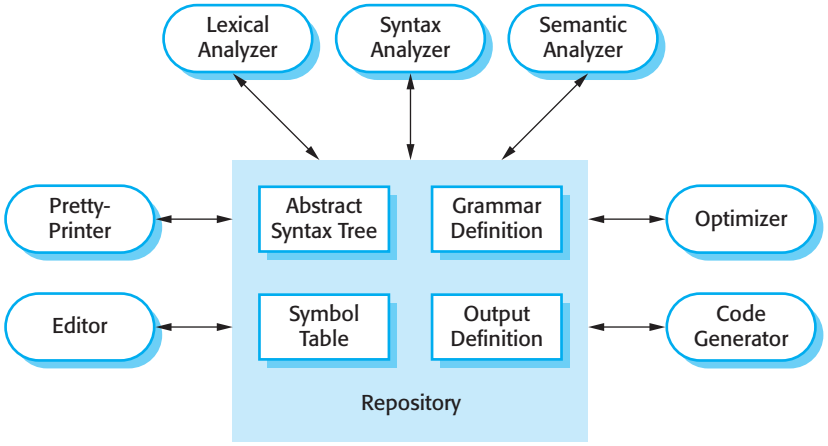


Figure 6.20 A repository architecture for a language processing system

KEY POINTS

- A software architecture is a description of how a software system is organized. Properties of a system such as performance, security, and availability are influenced by the architecture used.
- Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used, and the ways in which the architecture should be documented and evaluated.
- Architectures may be documented from several different perspectives or views. Possible views include a conceptual view, a logical view, a process view, a development view, and a physical view.
- Architectural patterns are a means of reusing knowledge about generic system architectures. They describe the architecture, explain when it may be used, and discuss its advantages and disadvantages.
- Commonly used architectural patterns include Model-View-Controller, Layered Architecture, Repository, Client–server, and Pipe and Filter.
- Generic models of application systems architectures help us understand the operation of applications, compare applications of the same type, validate application system designs, and assess large-scale components for reuse.
- Transaction processing systems are interactive systems that allow information in a database to be remotely accessed and modified by a number of users. Information systems and resource management systems are examples of transaction processing systems.
- Language processing systems are used to translate texts from one language into another and to carry out the instructions specified in the input language. They include a translator and an abstract machine that executes the generated language.

FURTHER READING

Software Architecture: Perspectives on an Emerging Discipline. This was the first book on software architecture and has a good discussion on different architectural styles. (M. Shaw and D. Garlan, Prentice-Hall, 1996.)

Software Architecture in Practice, 2nd ed. This is a practical discussion of software architectures that does not oversell the benefits of architectural design. It provides a clear business rationale explaining why architectures are important. (L. Bass, P. Clements and R. Kazman, Addison-Wesley, 2003.)

'The Golden Age of Software Architecture' This paper surveys the development of software architecture from its beginnings in the 1980s through to its current usage. There is little technical content but it is an interesting historical overview. (M. Shaw and P. Clements, *IEEE Software,* **21** (2), March–April 2006.) http://dx.doi.org/10.1109/MS.2006.58.

Handbook of Software Architecture. This is a work in progress by Grady Booch, one of the early evangelists for software architecture. He has been documenting the architectures of a range of software systems so you can see reality rather than academic abstraction. Available on the Web and intended to appear as a book. http://www.handbookofsoftwarearchitecture.com/.

EXERCISES

6.1. When describing a system, explain why you may have to design the system architecture before the requirements specification is complete.

6.2. You have been asked to prepare and deliver a presentation to a non-technical manager to justify the hiring of a system architect for a new project. Write a list of bullet points setting out the key points in your presentation. Naturally, you have to explain what is meant by system architecture.

6.3. Explain why design conflicts might arise when designing an architecture for which both availability and security requirements are the most important non-functional requirements.

6.4. Draw diagrams showing a conceptual view and a process view of the architectures of the following systems:

An automated ticket-issuing system used by passengers at a railway station.

A computer-controlled video conferencing system that allows video, audio, and computer data to be visible to several participants at the same time.

A robot floor cleaner that is intended to clean relatively clear spaces such as corridors. The cleaner must be able to sense walls and other obstructions.

6.5. Explain why you normally use several architectural patterns when designing the architecture of a large system. Apart from the information about patterns that I have discussed in this chapter, what additional information might be useful when designing large systems?

6.6. Suggest an architecture for a system (such as iTunes) that is used to sell and distribute music on the Internet. What architectural patterns are the basis for this architecture?

6.7. Explain how you would use the reference model of CASE environments (available on the book's web pages) to compare the IDEs offered by different vendors of a programming language such as Java.

6.8. Using the generic model of a language processing system presented here, design the architecture of a system that accepts natural language commands and translates these into database queries in a language such as SQL.

6.9. Using the basic model of an information system, as presented in Figure 6.16, suggest the components that might be part of an information system that allows users to view information about flights arriving and departing from a particular airport.

6.10. Should there be a separate profession of 'software architect' whose role is to work independently with a customer to design the software system architecture? A separate software company would then implement the system. What might be the difficulties of establishing such a profession?

REFERENCES

Bass, L., Clements, P. and Kazman, R. (2003). Software Architecture in Practice, 2nd ed. Boston: Addison-Wesley.

Berczuk, S. P. and Appleton, B. (2002). Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Boston: Addison-Wesley.

Booch, G. (2009). 'Handbook of software architecture'. Web publication. http://www.handbookofsoftwarearchitecture.com/.

Bosch, J. (2000). Design and Use of Software Architectures. Harlow, UK: Addison-Wesley.

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.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J. (2002). Documenting Software Architectures: Views and Beyond. Boston: Addison-Wesley.

Coplien, J. H. and Harrison, N. B. (2004). Organizational Patterns of Agile Software Development. 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.

Garlan, D. and Shaw, M. (1993). 'An introduction to software architecture'. Advances in Software Engineering and Knowledge Engineering, 1 1–39.

Harold, E. R. and Means, W. S. (2002). XML in a Nutshell. Sebastopol. Calif.: O'Reilly.

Hofmeister, C., Nord, R. and Soni, D. (2000). Applied Software Architecture. Boston: Addison-Wesley.

Hunter, D., Rafter, J., Fawcett, J. and Van Der Vlist, E. (2007). Beginning XML, 4th ed. Indianapolis, Ind.: Wrox Press.

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

Krutchen, P. (1995). 'The 4+1 view model of software architecture'. IEEE Software, 12 (6), 42–50.

Lange, C. F. J., Chaudron, M. R. V. and Muskens, J. (2006). 'UML software description and architecture description'. IEEE Software, 23 (2), 40–6.

Lewis, P. M., Bernstein, A. J. and Kifer, M. (2003). Databases and Transaction Processing: An Application-oriented Approach. Boston: Addison-Wesley.

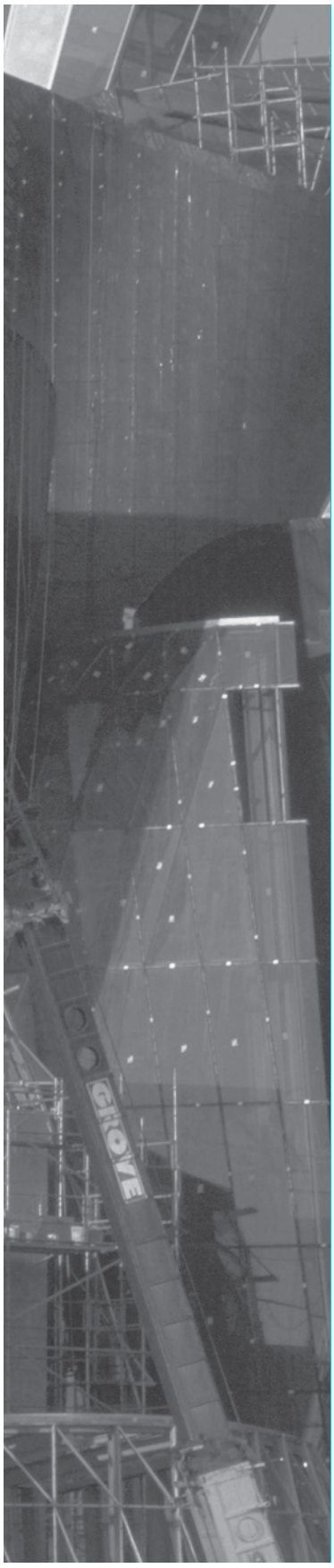
Martin, D. and Sommerville, I. (2004). 'Patterns of interaction: Linking ethnomethodology and design'. ACM Trans. on Computer-Human Interaction, 11 (1), 59–89.

Nii, H. P. (1986). 'Blackboard systems, parts 1 and 2'. AI Magazine, 7 (3 and 4), 38–53 and 62–9.

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.

Shaw, M. and Garlan, D. (1996). Software Architecture: Perspectives on an Emerging Discipline. Englewood Cliffs, NJ: Prentice Hall.

Usability group. (1998). 'Usability patterns'. Web publication. http://www.it.bton.ac.uk/cil/usability/patterns/.



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 design and implementation is software engineering, and all other activities are merged with this process. However, for large systems, software design and implementation is only one of a set of processes (requirements engineering, verification and validation, etc.) involved in software engineering.

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 in a dynamically typed language like Python and makes no sense at all 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 many 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 or not you should buy or build the application software. In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) 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 can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.

When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements. You don't usually develop design models of the system, such as models of the system objects and their interactions. I discuss this COTS-based approach to development in Chapter 16.

I assume that most readers of this book will 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.

Structured design methods

Structured design methods propose that software design should be tackled in a methodical way. Designing a system involves following the steps of the method and refining the design of a system at increasingly detailed levels. In the 1990s, there were a number of competing methods for object-oriented design. However, the inventors of the most commonly used methods came together and invented the UML, which unified the notations used in the different methods.

Rather than focus on methods, most discussions now are about processes where design is seen as part of the overall software development process. The Rational Unified Process (RUP) is a good example of a generic development process.

http://www.SoftwareEngineering-9.com/Web/Structured-methods/

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 towards any particular programming language or implementation technology. Therefore, I have presented all examples using the UML rather than in 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.

Object-oriented systems are easier to change than systems developed using functional approaches. 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, there are several things that you need to do:

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. Consequently, I have deliberately not illustrated this process as a simple diagram because that would imply design can be thought of as a neat sequence of activities. In fact, all of the above activities are interleaved and so influence each other.

I illustrate these process activities by designing part of the software for the wilderness weather station that I introduced in Chapter 1. Wilderness weather stations are deployed in remote areas. Each 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. Understanding of 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. The nature of the relationships is now specified. You may therefore document the environment of the system using a simple block diagram, showing the entities in the system and their associations. This is illustrated in Figure 7.1, which shows that

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://www.SoftwareEngineering-9.com/Web/WS/Usecases.html

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 one 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 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. As I explained in Chapter 5, a 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

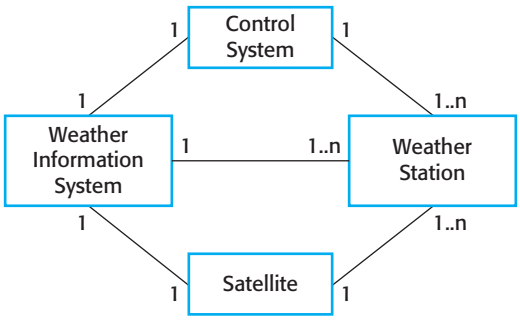


Figure 7.1 System context for the weather station

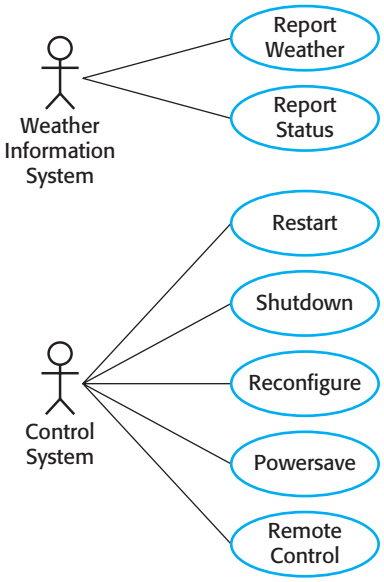


Figure 7.2 Weather station use cases

so on. This is shown in Figure 7.3, which describes the Report weather use case from Figure 7.2. Examples of some other use cases are on the Web.

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 with your general knowledge of the principles of architectural design and with more detailed domain knowledge.

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Dat	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 five-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 are 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 the future.

Figure 7.3 Use case description—Report weather

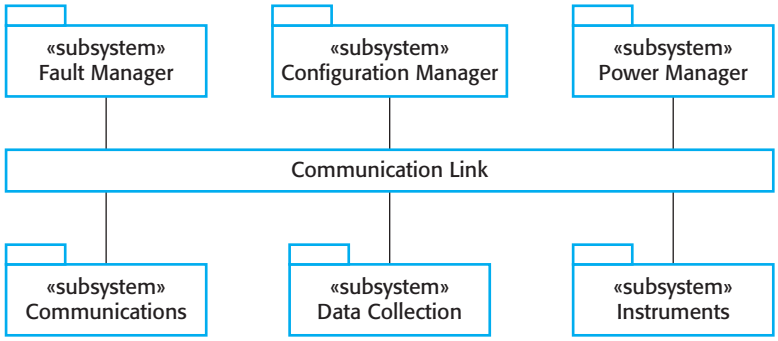


Figure 7.4 High-level architecture of the weather station

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

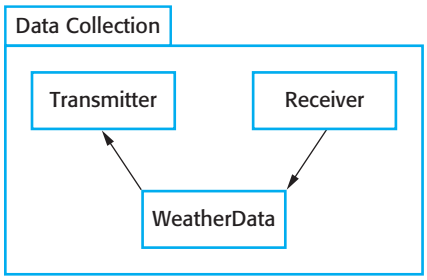


Figure 7.5 Architecture of data collection system

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

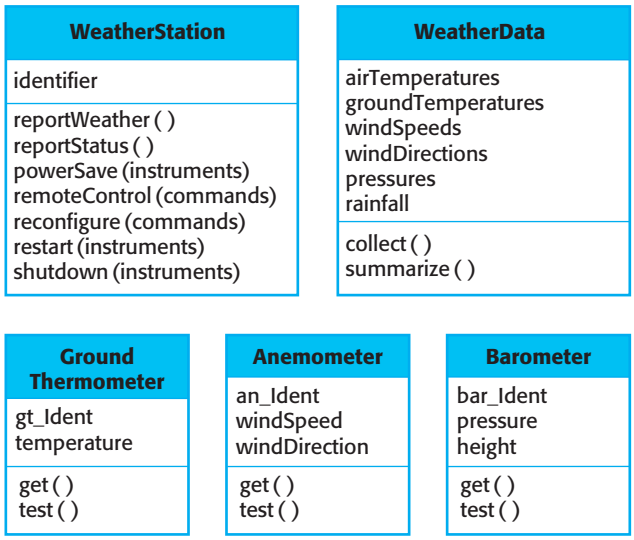


Figure 7.6 Weather station objects

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.

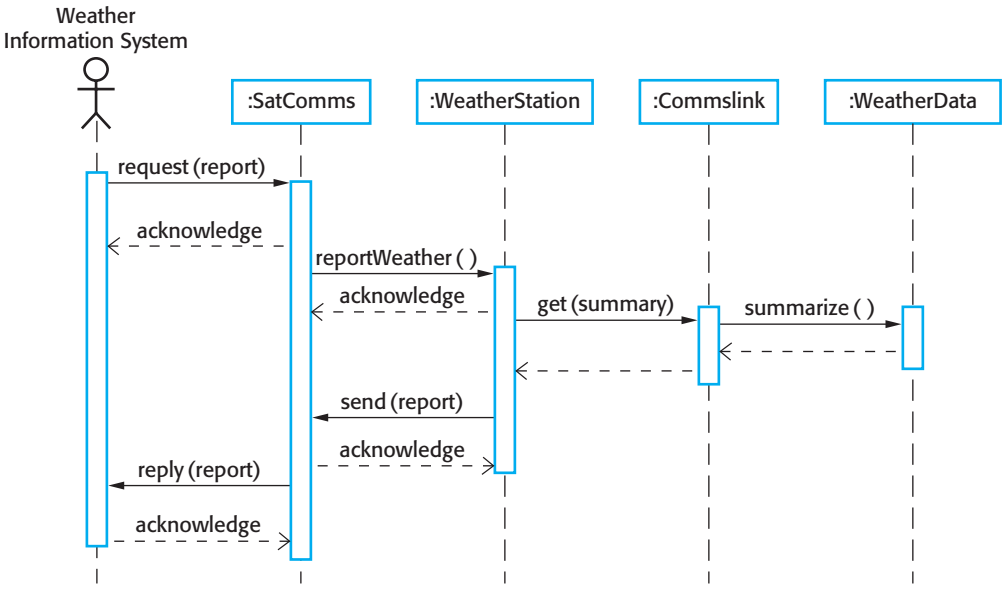


Figure 7.7 Sequence diagram describing data collection

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.

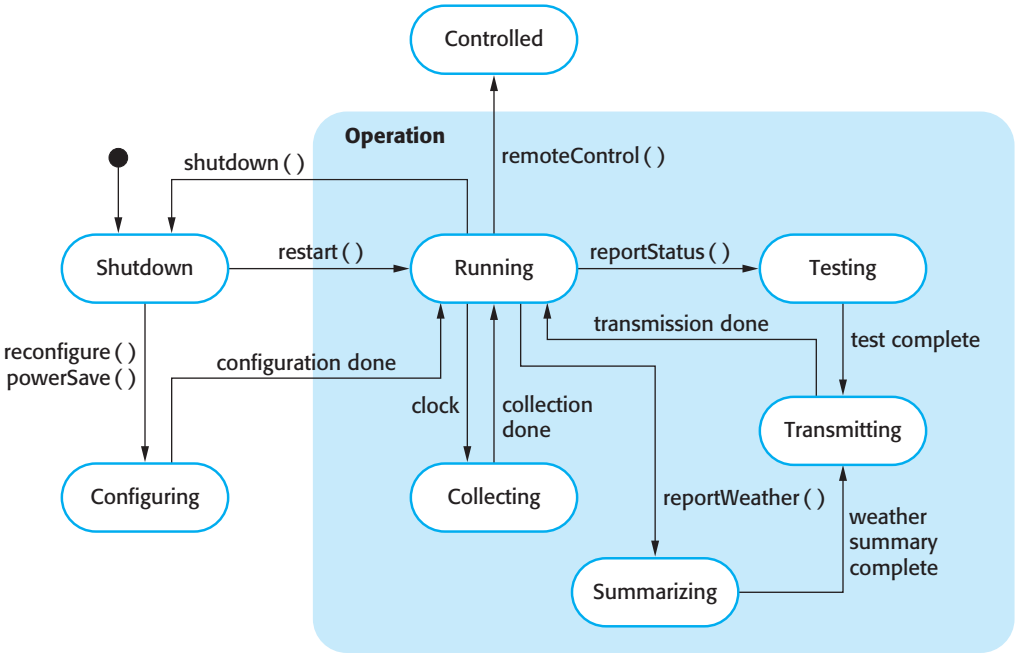


Figure 7.8 Weather station state diagram

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 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

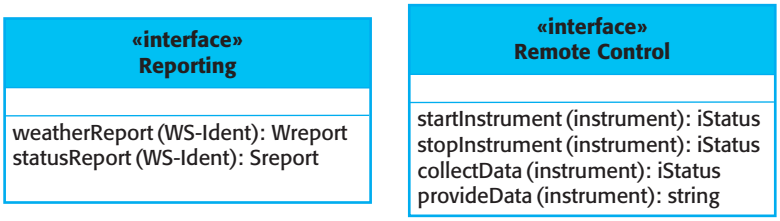


Figure 7.9 Weather station interfaces

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

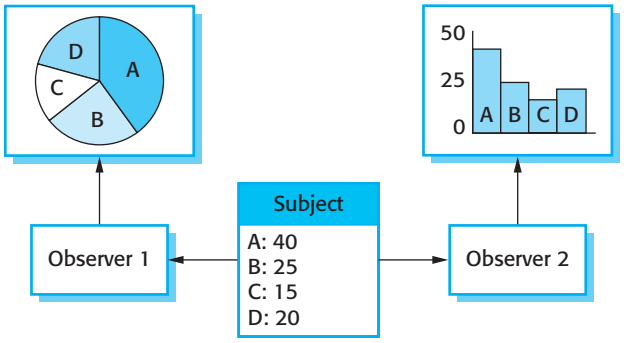


Figure 7.11 Multiple displays

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

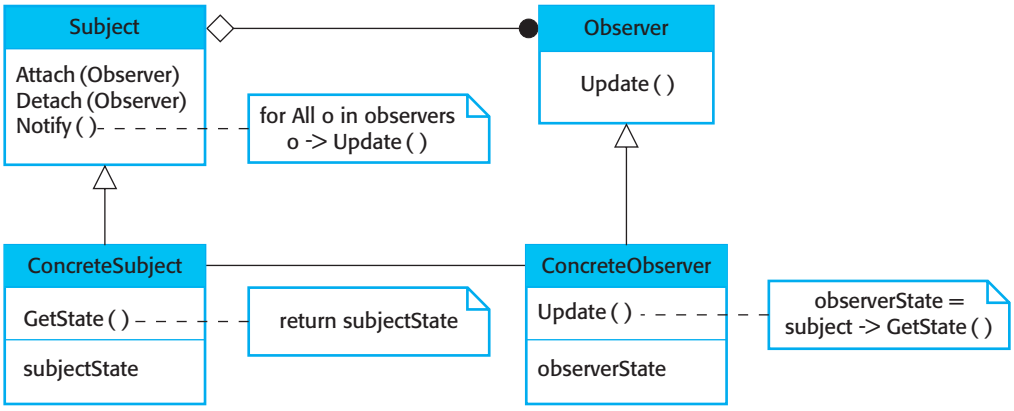


Figure 7.12 A UML model of the Observer pattern

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.