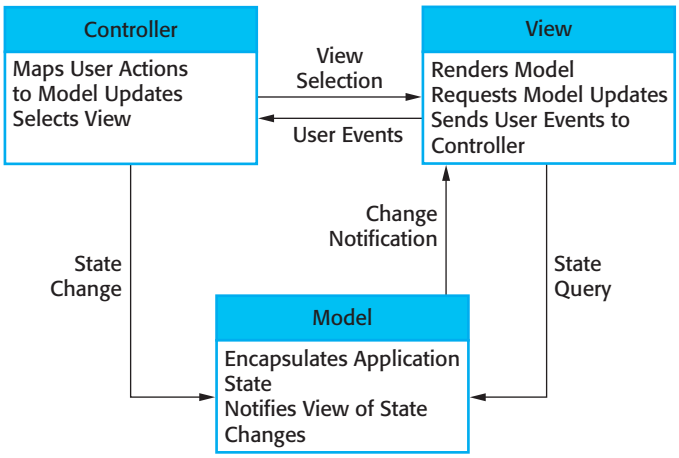


Figure 6.3 The organization of the MVC

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.

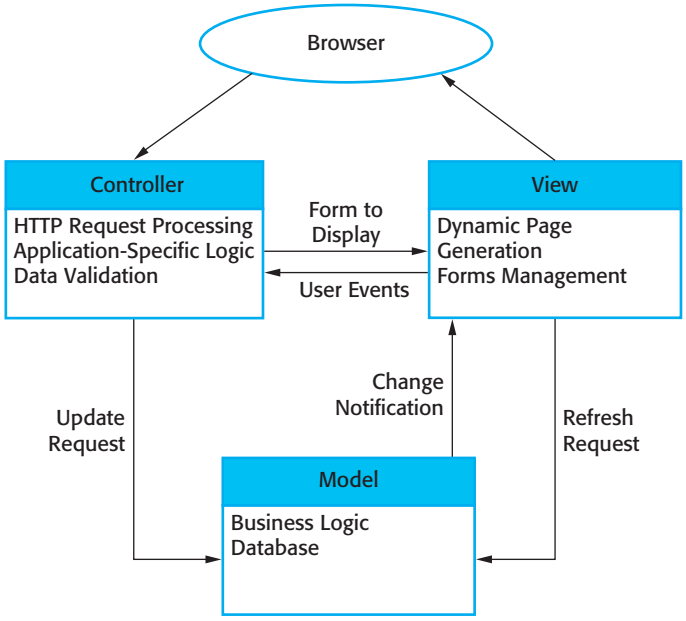


Figure 6.4 Web application architecture using the MVC pattern

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.

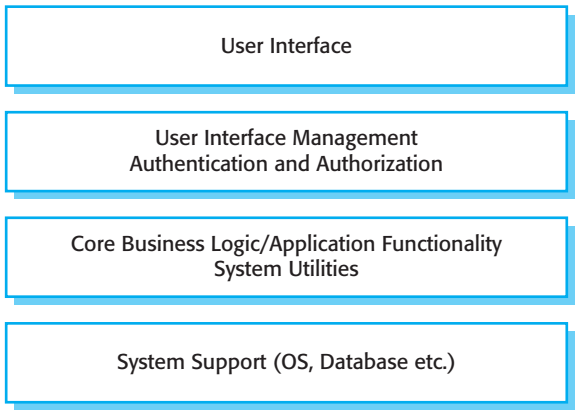


Figure 6.6 A generic layered architecture

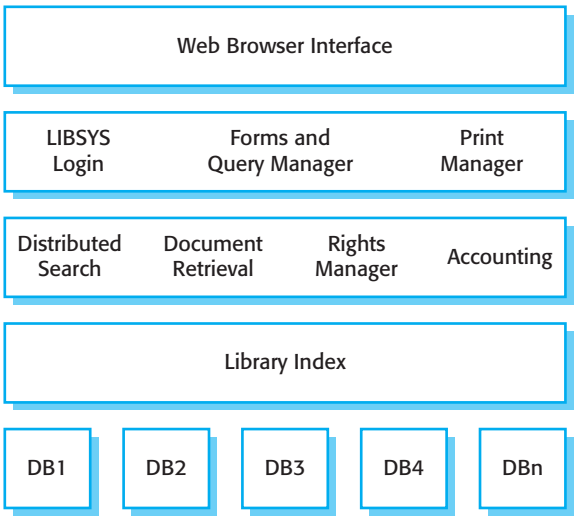


Figure 6.7 The architecture of the LIBSYS system

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.

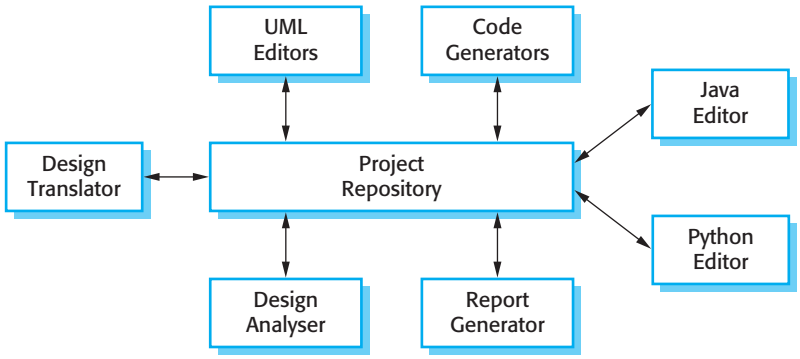


Figure 6.9 A repository architecture for an IDE

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

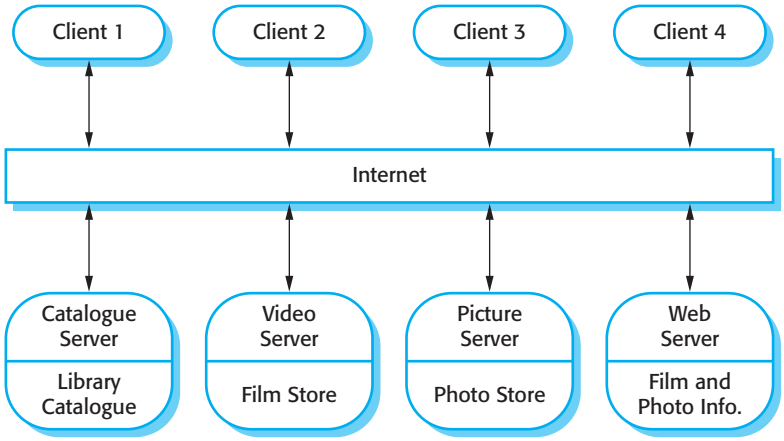


Figure 6.11 A client–server architecture for a film library

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.

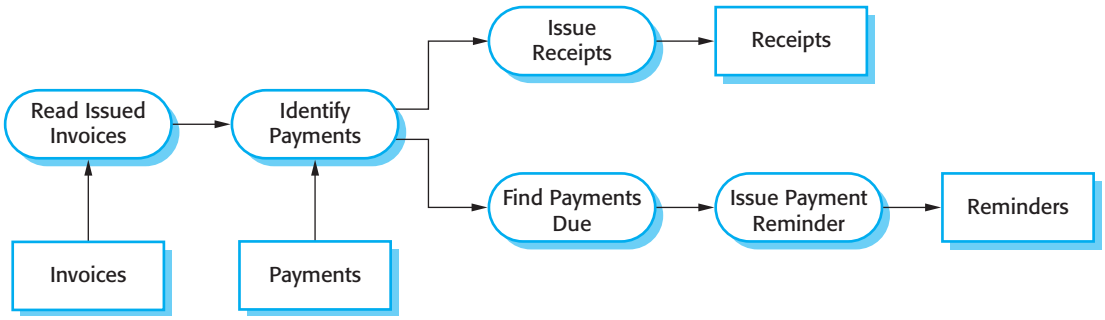


Figure 6.13 An example of the pipe and filter architecture

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

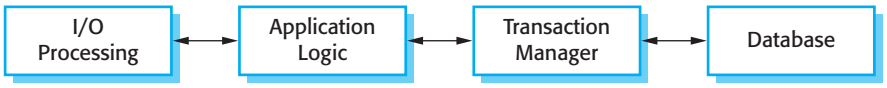


Figure 6.14 The structure of transaction processing applications

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

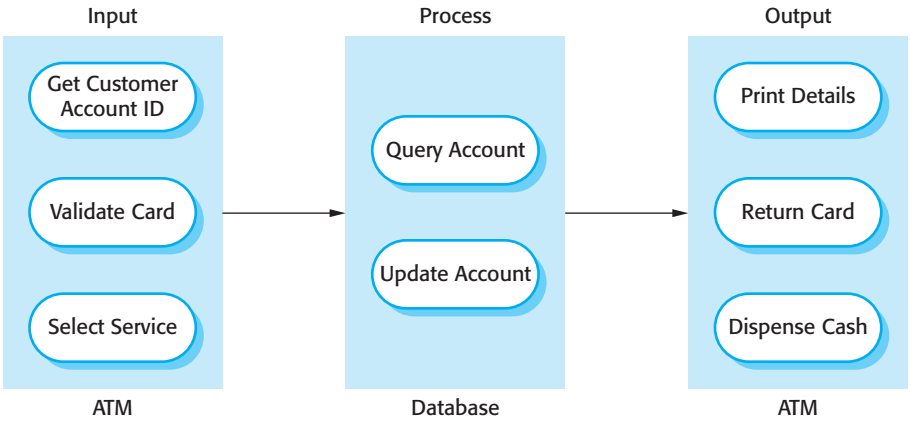


Figure 6.15 The software architecture of an ATM system

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

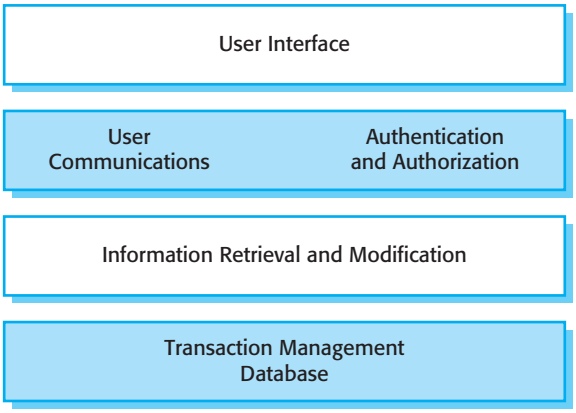


Figure 6.16 Layered information system architecture

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.

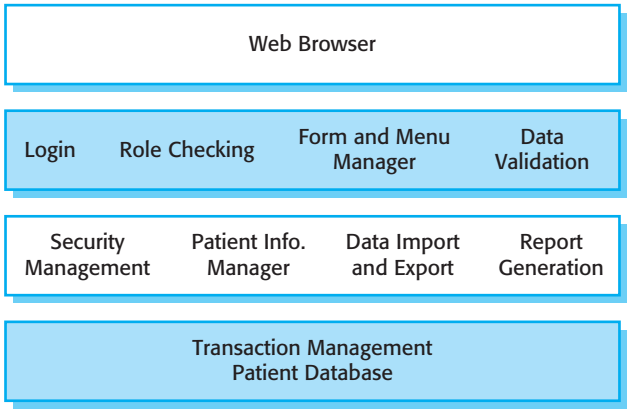


Figure 6.17 The architecture of the MHC-PMS

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.

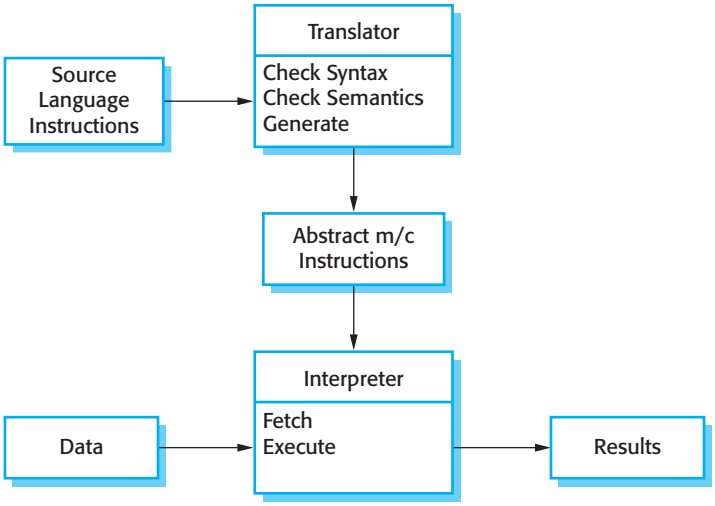


Figure 6.18 The architecture of a language processing system

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.

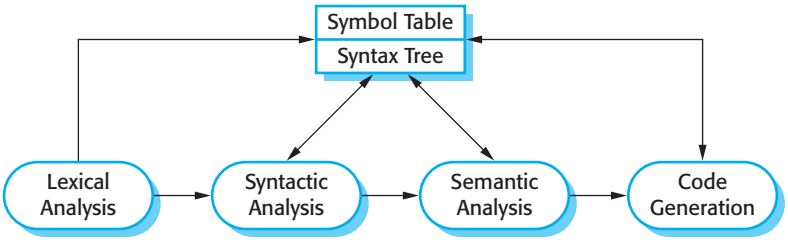


Figure 6.19 A pipe and filter compiler architecture

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.

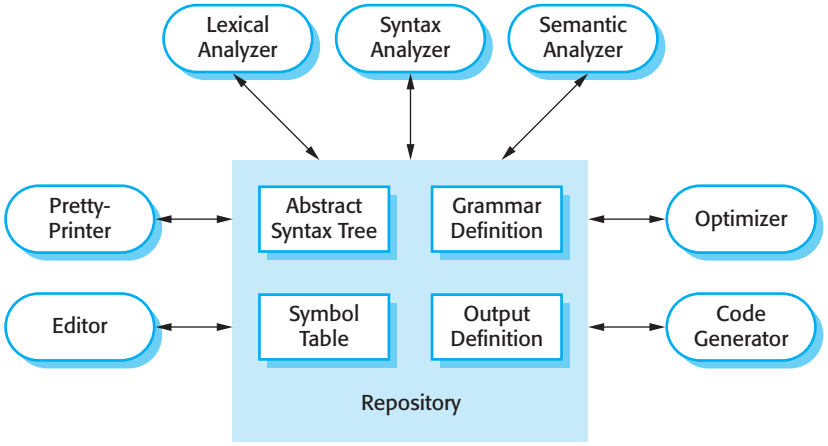


Figure 6.20 A repository architecture for a language processing system

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.

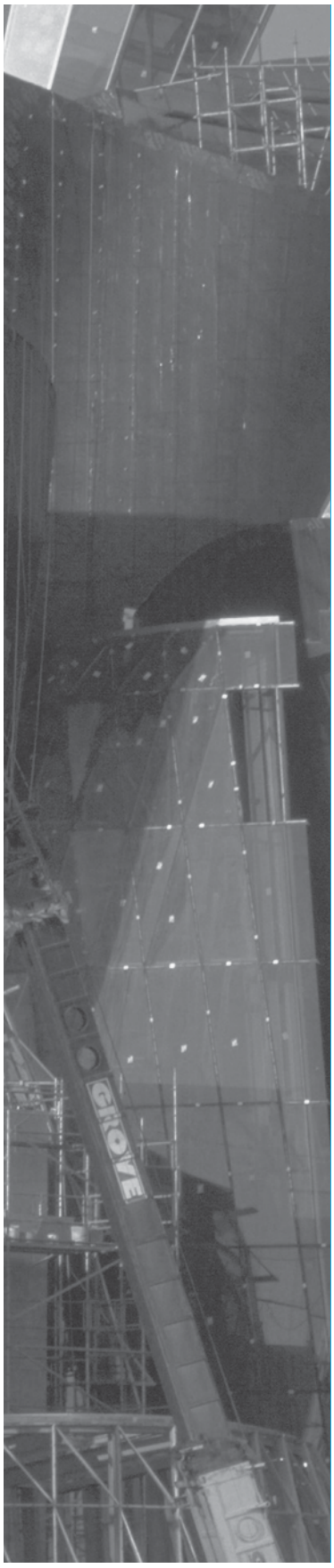
Martin, D. and Sommerville, I. (2004). 'Patterns of interaction: Linking ethnomethodology and design'. ACM Trans. on Computer-Human Interaction, 11 (1), 59–89.

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

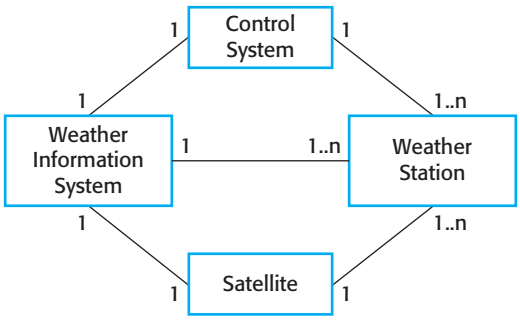


Figure 7.1 System context for the weather station

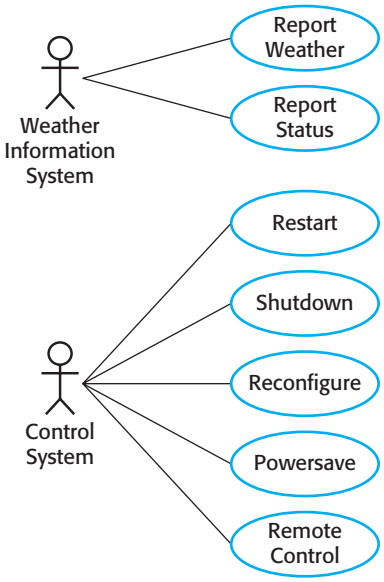


Figure 7.2 Weather station use cases

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