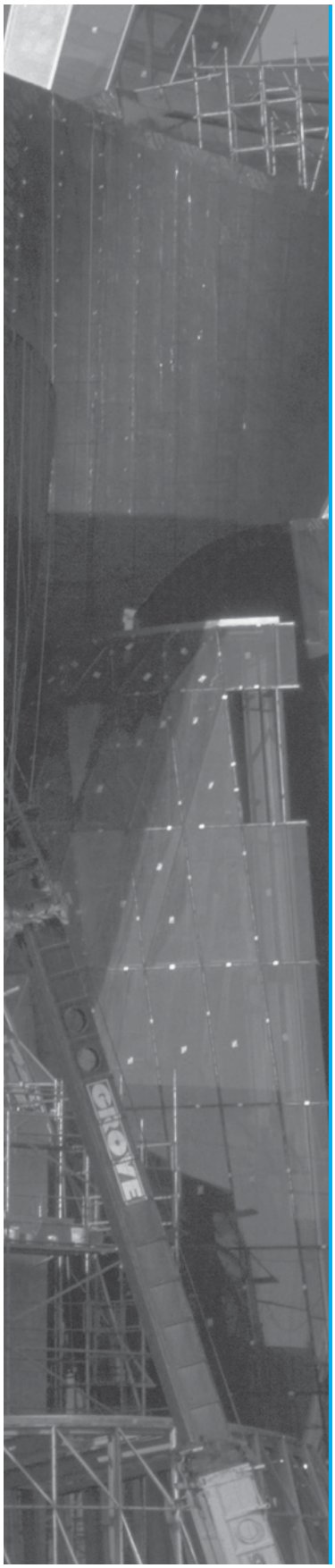

6 Architectural design

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.

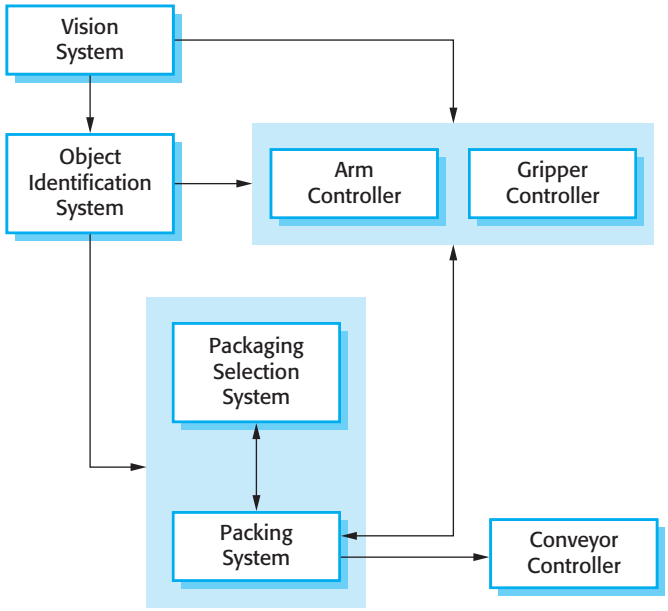

Figure 6.1 The architecture of a packing robot control system

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

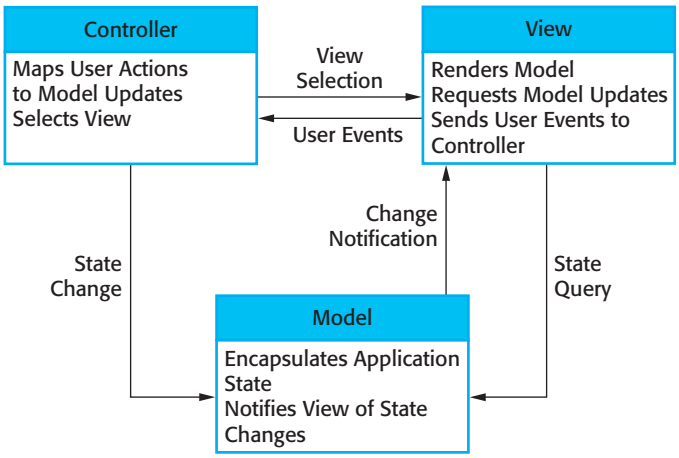

Figure 6.3 The organization of the MVC

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.

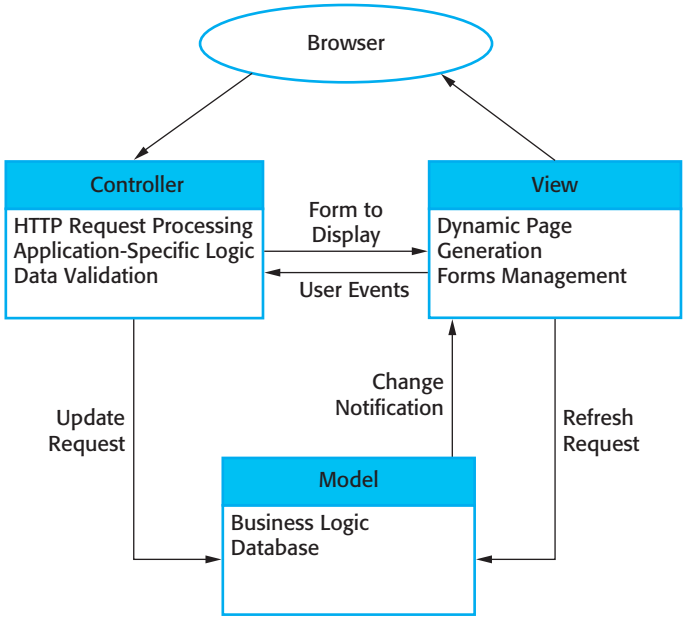

Figure 6.4 Web application architecture using the MVC pattern

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.

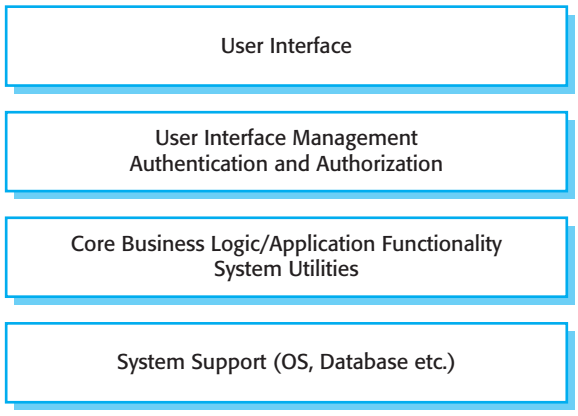

Figure 6.6 A generic layered architecture

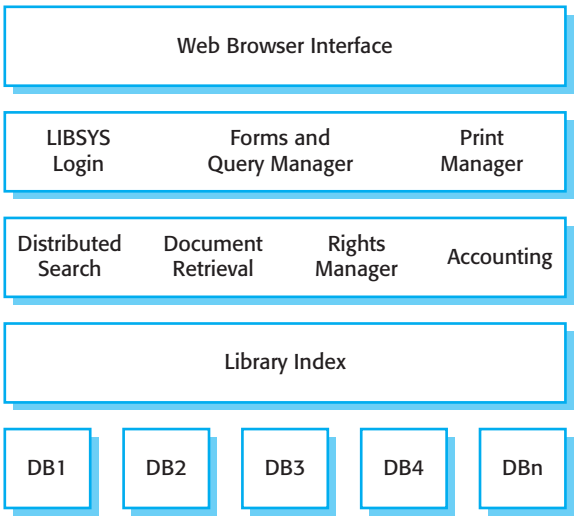

Figure 6.7 The architecture of the LIBSYS system

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.

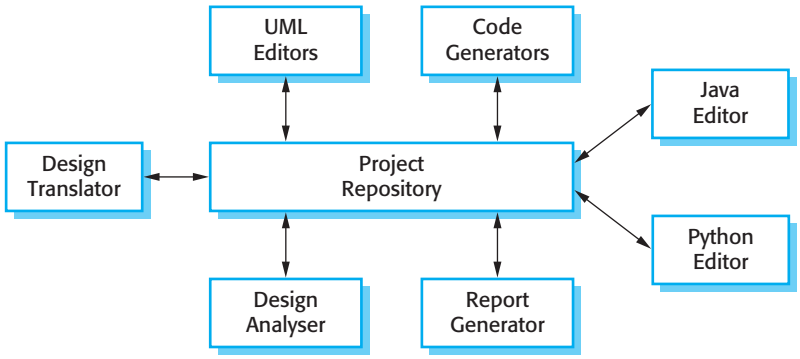

Figure 6.9 A repository architecture for an IDE

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

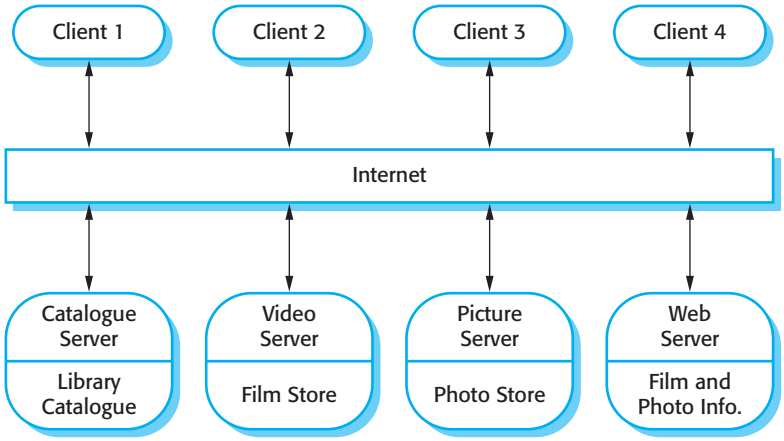

Figure 6.11 A client–server architecture for a film library

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.

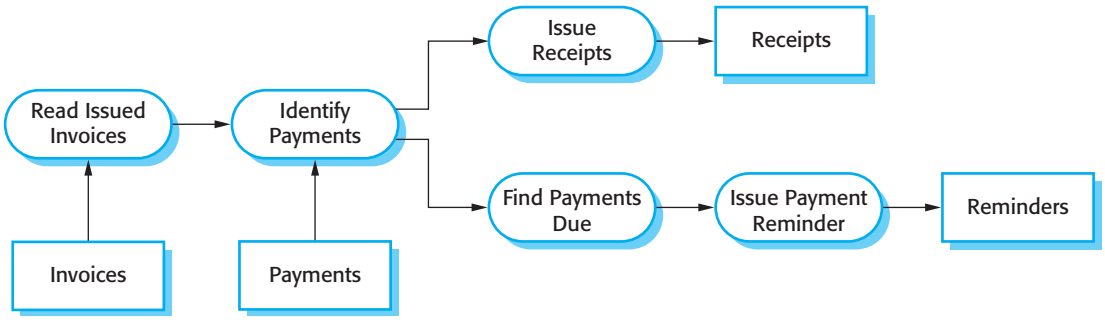

Figure 6.13 An example of the pipe and filter architecture

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

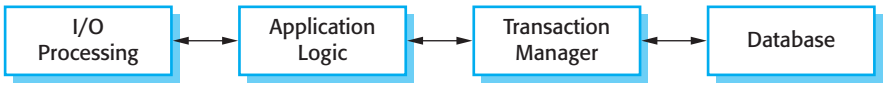

Figure 6.14 The structure of transaction processing applications

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

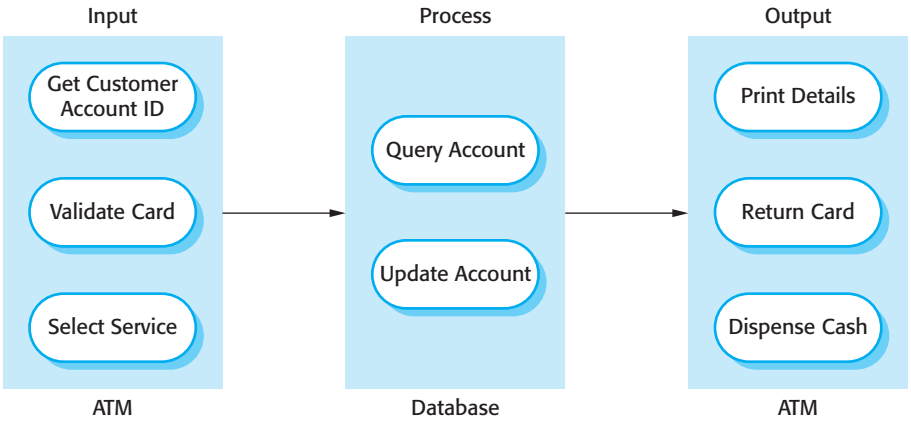

Figure 6.15 The software architecture of an ATM system

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

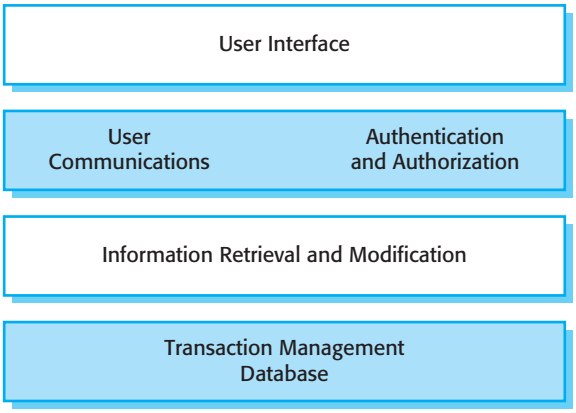

Figure 6.16 Layered information system architecture

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.

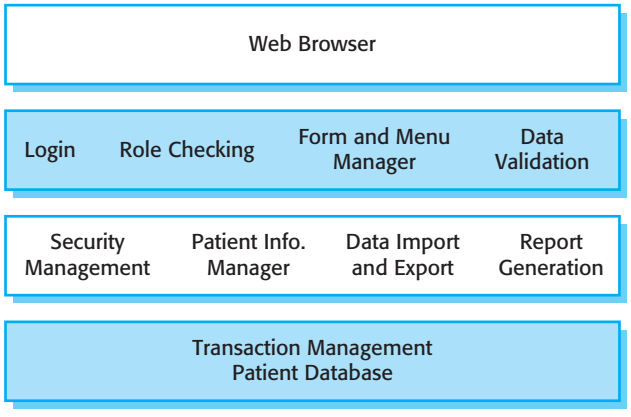

Figure 6.17 The architecture of the MHC-PMS

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.

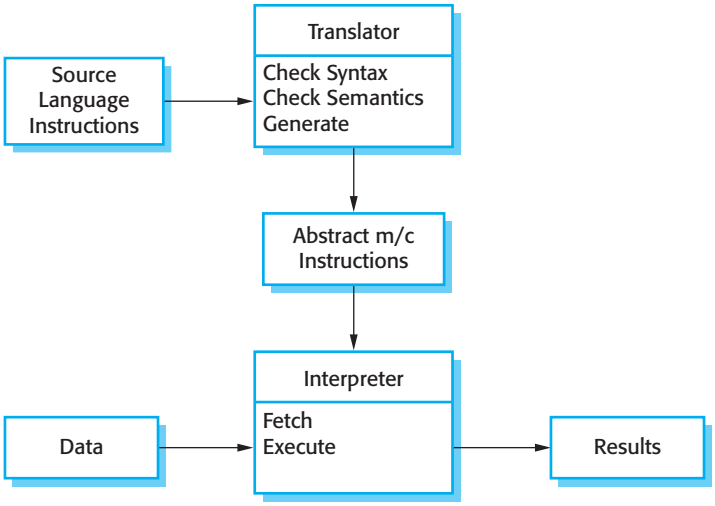

Figure 6.18 The architecture of a language processing system

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.

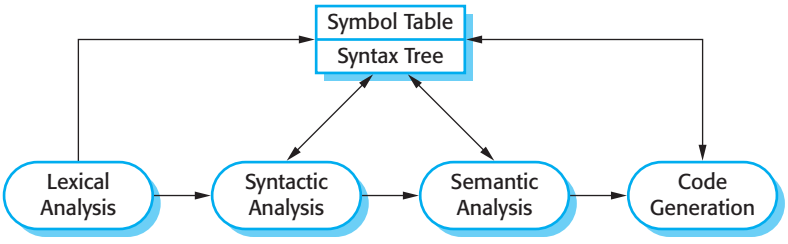

Figure 6.19 A pipe and filter compiler architecture

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.

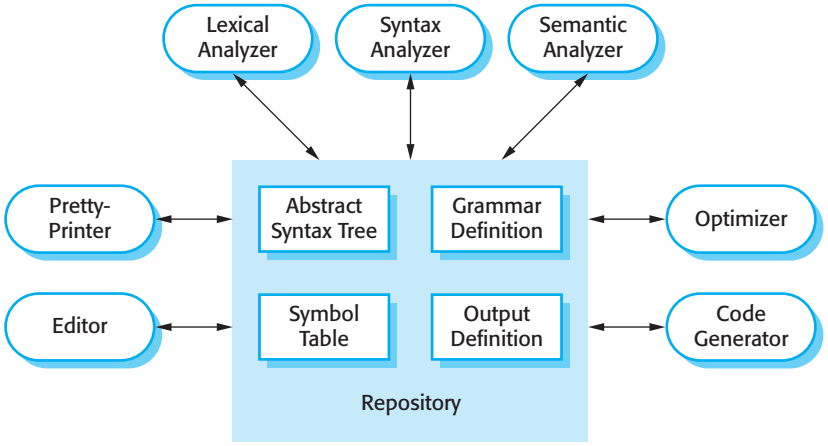

Figure 6.20 A repository architecture for a language processing system

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.