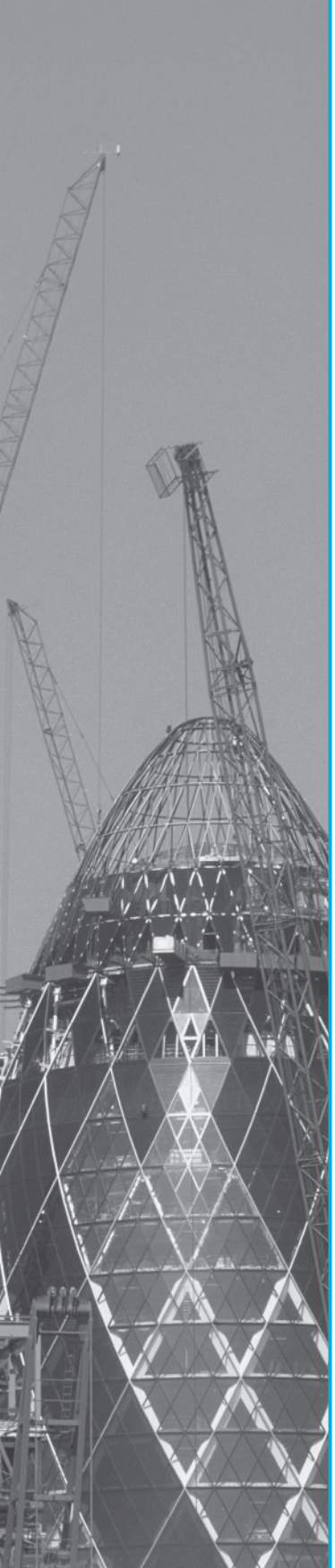


6 Architectural design

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 software architecture during the architectural design process;
- have been introduced to the idea of Architectural patterns, well-tried ways of organizing software architectures that can be reused in system designs;
- understand how Application-Specific Architectural patterns may be used in transaction processing and language processing systems.

Contents

Architectural design is concerned with understanding how a software system should be organized and designing the overall structure of that system. In the model of the software development process that I described 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 an agile development process should focus on designing an overall system architecture. Incremental development of architectures is not usually successful. Refactoring components in response to changes is usually relatively easy. However, refactoring the system architecture is expensive because you may need to modify most system components to adapt them to the architectural changes.

To help you understand what I mean by system architecture, look at Figure 6.1. This diagram shows an abstract model of the architecture for a packing robot system. This robotic system can pack different kinds of objects. 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

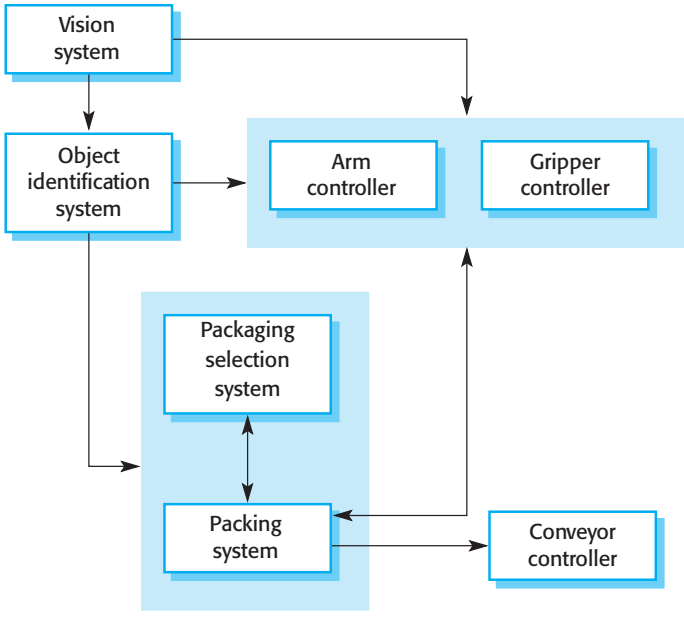


Figure 6.1 The architecture of a packing robot control system

include any design information. This ideal is unrealistic, however, except for very small systems. You need to identify the main architectural components as these reflect the high-level features of the system. 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 subsystems. You then use this decomposition to discuss the requirements and more detailed 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 may be distributed over different computers, which may be owned and managed by different companies. (I cover architecture in the large in Chapters 17 and 18.)

Software architecture is important because it affects the performance, robustness, distributability, and maintainability of a system (Bosch 2000). As Bosch explains, individual components implement the functional system requirements, but the dominant influence on the non-functional system characteristics is the system's architecture. Chen et al. (Chen, Ali Babar, and Nuseibeh 2013) confirmed this in a study of "architecturally significant requirements" where they found that non-functional requirements had the most significant effect on the system's architecture.

Bass et al. (Bass, Clements, and Kazman 2012) suggest that explicitly designing and documenting software architecture has three advantages:

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* An architectural model 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 15, product-line architectures are an approach to reuse where the same architecture is reused across a range of related systems.

System architectures are often modeled informally 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 subcomponents. 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 handbook of software architecture (Booch 2014).

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. In spite of their widespread use, Bass et al. (Bass, Clements, and Kazman 2012) 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 architectural theory and industrial practice arise because there are two ways in which an architectural model of a program is used:

1. *As a way of encouraging discussions 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 that 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 such a model is that such a detailed architectural description makes it easier to understand and evolve the system.

Block diagrams are a good way of supporting communications between the people involved in the software design process. They are intuitive, and domain experts and software engineers can relate to them and participate in discussions about the system. Managers find them helpful in planning the project. For many projects, block diagrams are the only architectural description.

Ideally, if the architecture of a system is to be documented in detail, it is better to use a more rigorous notation for architectural description. Various architectural description languages (Bass, Clements, and Kazman 2012) have been developed for this purpose. A more detailed and complete description means that there is less scope for misunderstanding the relationships between the architectural components. However, developing a detailed architectural description is an expensive and time-consuming process. It is practically impossible to know whether or not it is cost-effective, so this approach is not widely used.

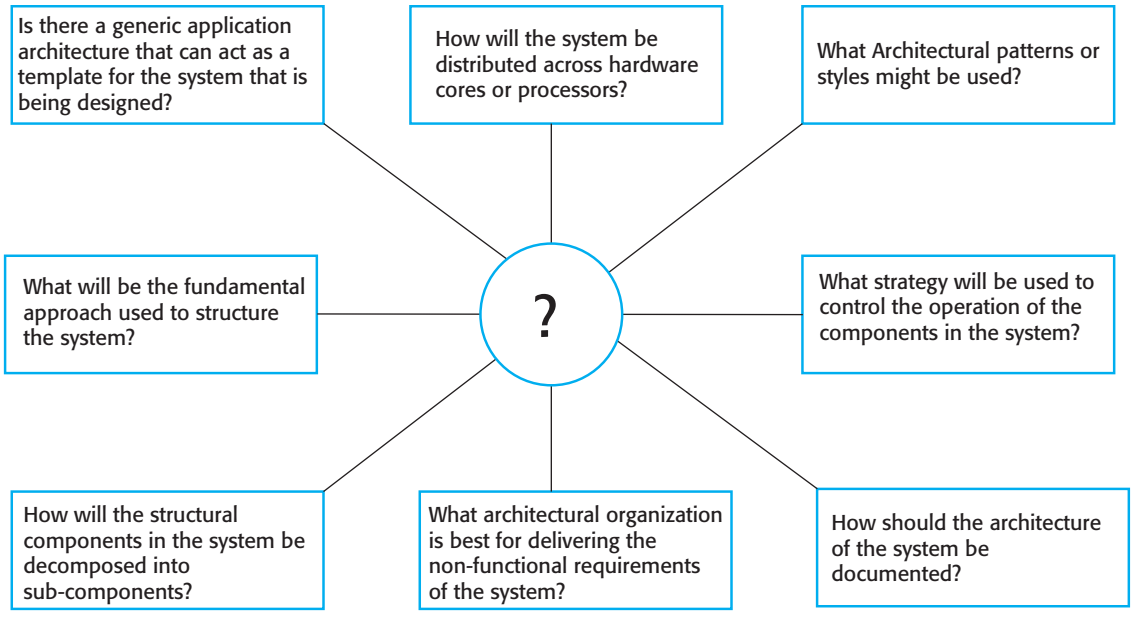


Figure 6.2 Architectural design decisions

6.1 Architectural design decisions

Architectural design is a creative process in which you design a system organization that will satisfy the functional and non-functional requirements of a system. There is no formulaic architectural design process. It depends on the type of system being developed, the background and experience of the system architect, and the specific requirements for the system. Consequently, I think it is best to consider 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 fundamental questions shown in Figure 6.2.

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.

For embedded systems and apps designed for personal computers and mobile devices, you do not have to design a distributed architecture for the system. However, most large systems are 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 that I cover in Chapter 17.

The architecture of a software system may be based on a particular Architectural pattern or style (these terms have come to mean the same thing). 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 cover several frequently used patterns in Section 6.3.

Garlan and Shaw's notion of an architectural style covers questions 4 to 6 in the list of fundamental architectural questions shown in Figure 6.2. 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 a strategy for decomposing components into subcomponents. Finally, in the control modeling process, you develop a general model of the control relationships between the various parts of the system and make decisions about how the execution of components is controlled.

Because of the close relationship between non-functional system characteristics and software architecture, the choice of architectural style and structure should depend on the non-functional requirements of the system:

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 deployed on the same computer rather than distributed across the network. This may mean using a few relatively large components rather than small, finer-grain components. Using large components reduces the number of component communications, as most of the interactions between related system features take place within a component. You may also consider runtime 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 and 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 co-located in a single component or in a small number of components. This reduces the costs and problems of safety validation and may make 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 fault-tolerant system architectures for high-availability systems in Chapter 11.

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, however, then some compromise must be found. You can sometimes do this by using different Architectural patterns or styles for separate parts of the system. Security is now almost always a critical requirement, and you have to design an architecture that maintains security while also satisfying other non-functional requirements.

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 description (Bosch 2000) of the non-functional characteristics of some Architectural patterns can 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 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 diagram, as a graphical model can only show one view or perspective of the system. It might show how a system is decomposed into modules, how the runtime processes interact, or the different ways in which system components are distributed across a network. Because all of these are useful at different times, 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 (Krutchen 1995) in his well-known 4+1 view model of software architecture, suggests that there should

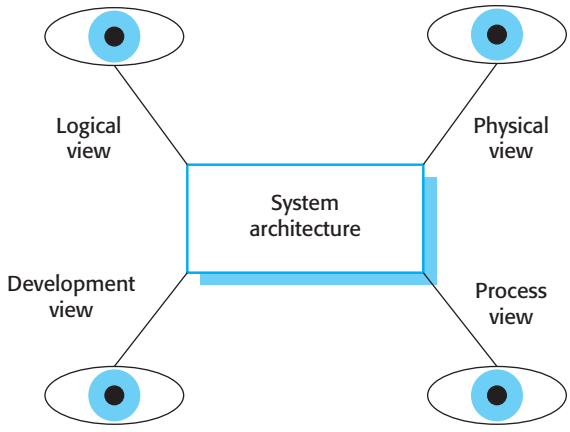


Figure 6.3 Architectural views

be four fundamental architectural views, which can be linked through common use cases or scenarios (Figure 6.3). He suggests the following views:

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 runtime, 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. (Hofmeister, Nord, and Soni 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 15) 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 of a system's architecture are almost always developed during the design process. They are used to explain the system architecture to stakeholders and to inform architectural decision making. During the design process, some of the other views may also be developed when different aspects of the system are discussed, but it is rarely necessary to develop 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 describing and documenting software architectures. A survey in 2006 (Lange, Chaudron, and Muskens 2006) showed that, when the UML was used, it was mostly applied in an 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 that 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 (Bass, Clements, and Kazman 2012) have proposed the use of more specialized architectural description languages (ADLs) 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 ADLs are specialist languages, domain and application specialists find it hard to understand and use ADLs. There may be some value in using domain-specific ADLs as part of model-driven development, but I do not think they will become part of mainstream software engineering practice. 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 these documents. I largely agree with this view, and I think that, except for critical systems, it is not worth developing a detailed architectural description from Krutchen's four perspectives. You should develop the views that are useful for communication and not worry about whether or not your architectural documentation is complete.

6.3 Architectural patterns

The idea of patterns as a way of presenting, sharing, and reusing knowledge about software systems has been adopted in a number of areas of software engineering. The trigger for this was the publication of a book on object-oriented design patterns (Gamma et al. 1995). This prompted the development of other types of patterns, such as patterns for organizational design (Coplien and Harrison 2004), usability patterns (Usability Group 1998), patterns of cooperative interaction (Martin and Sommerville 2004), and configuration management patterns (Berczuk and Appleton 2002).

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

In this section, I introduce Architectural patterns and briefly describe a selection of Architectural patterns that are commonly used. Patterns may be described in a standard way (Figures 6.4 and 6.5) using a mixture of narrative description and diagrams.

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 6.5.
Example	Figure 6.6 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	May involve additional code and code complexity when the data model and interactions are simple.

Figure 6.4 The Model-View-Controller (MVC) pattern

For more detailed information about patterns and their use, you should refer to the 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 on when it is and is not appropriate to use that pattern, and details on the pattern's strengths and weaknesses.

Figure 6.4 describes the well-known Model-View-Controller pattern. This pattern is the basis of interaction management in many web-based systems and is supported by most language frameworks. The stylized pattern description includes the pattern

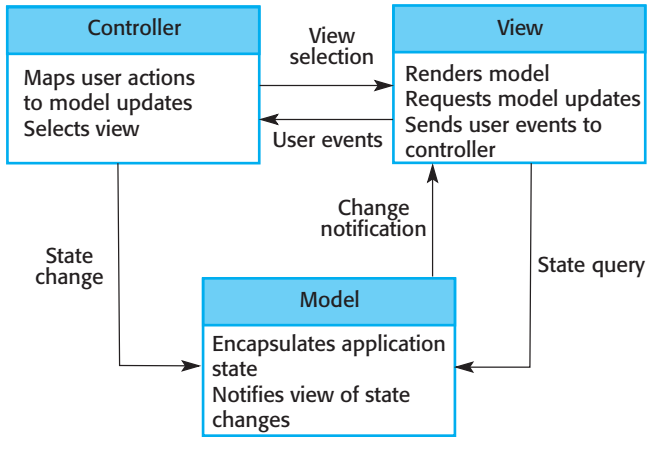


Figure 6.5 The organization of the Model-View-Controller

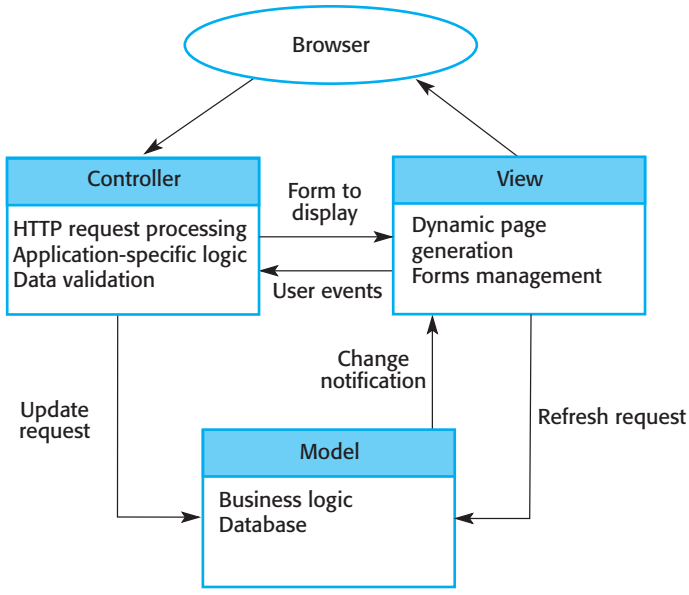


Figure 6.6 Web application architecture using the MVC pattern

name, a brief description, a graphical model, and an example of the type of system where the pattern is used. 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 6.5 and 6.6. These present the architecture from different views: Figure 6.5 is a conceptual view, and Figure 6.6 shows a runtime system architecture when this pattern is used for interaction management in a web-based system.

In this short space, it is impossible to describe all of the generic patterns that can be used in software development. Instead, I present some selected examples of patterns that are widely used and that capture good architectural design principles.

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 6.4, 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 6.7. 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. If its interface is unchanged, a new layer with extended functionality can replace an existing layer

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 6.8.
Example	A layered model of a digital learning system to support learning of all subjects in schools (Figure 6.9).
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 multilevel security.
Advantages	Allows replacement of entire layers as 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 6.7 The Layered Architecture pattern

without changing other parts of the system. 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, this makes it easier to provide multi-platform implementations of an application system. Only the machine-dependent layers need be reimplemented to take account of the facilities of a different operating system or database.

Figure 6.8 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, which includes the components concerned with the application functionality and utility components used by other application components.

The third layer is concerned with user interface 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.

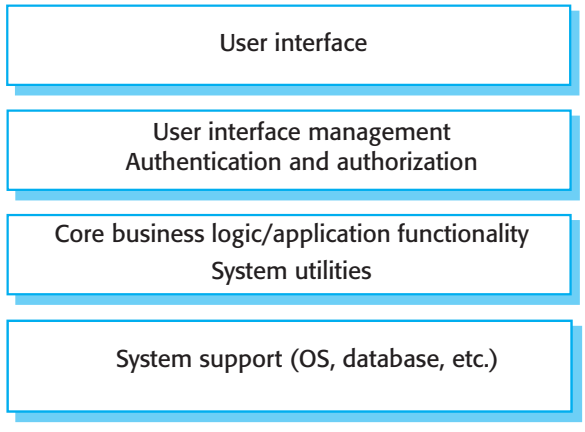


Figure 6.8 A generic layered architecture

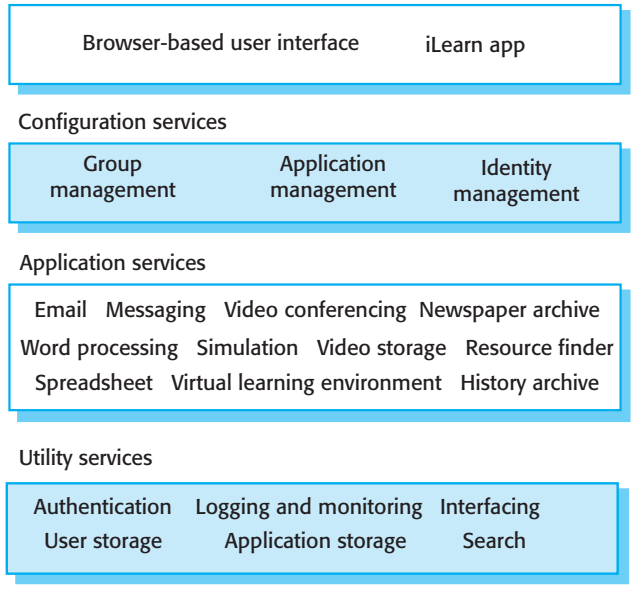


Figure 6.9 The architecture of the iLearn system

Figure 6.9 shows that the iLearn digital learning system, introduced in Chapter 1, has a four-layer architecture that follows this pattern. You can see another example of the Layered Architecture pattern in Figure 6.19 (Section 6.4, which shows the organization of the Mentcare 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 6.10), describes how a set of interacting components can share data.

Figure 6.10 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 6.11 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.

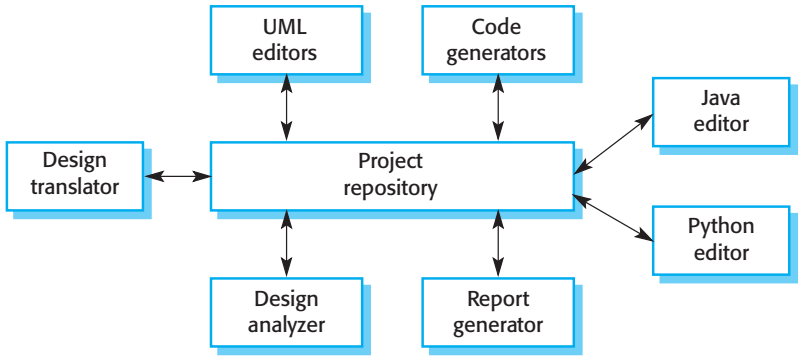


Figure 6.11 A repository architecture for an IDE

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

Figure 6.11 illustrates 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 rollback to earlier versions.

Organizing tools around a repository is an efficient way of sharing 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, this involves maintaining multiple copies of data. Keeping these consistent and up to date adds more overhead to the system.

In the repository architecture shown in Figure 6.11, the repository is passive and control is the responsibility of the components using the repository. An alternative approach, which has been derived for artificial intelligence (AI) systems, uses a "blackboard" model that triggers components when particular data become available. This is appropriate when the data in the repository is unstructured. Decisions about which tool is to be activated can only be made when the data has been analyzed. This model was introduced by Nii (Nii 1986), and Bosch (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 runtime organization. My next example, the Client–Server pattern (Figure 6.12), illustrates a commonly used runtime organization for distributed

Name	Client–server
Description	In a client–server architecture, the system is presented as a set of services, with each service delivered by a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.13 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 and so is susceptible to denial-of-service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. Management problems may arise if servers are owned by different organizations.

Figure 6.12 The Client–Server pattern

systems. 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 that offers programming language compilation services. Servers are software components, and several servers may run on the same computer.
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. Client–server systems are usually 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 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 http), where a client makes a request to a server and waits until it receives a reply from that server.

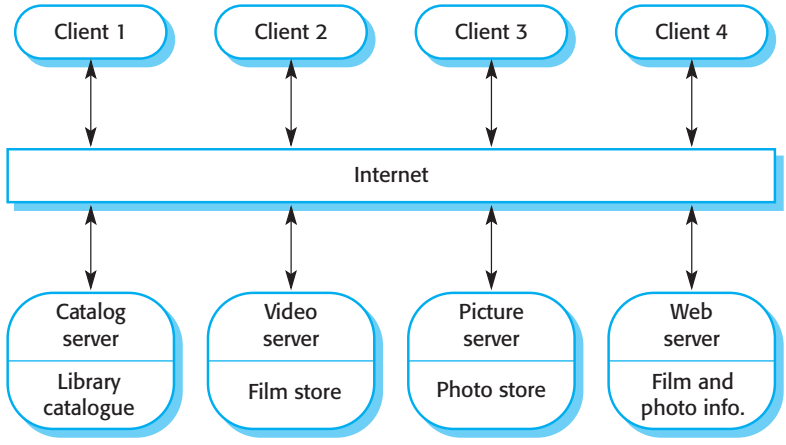


Figure 6.13 A client–server architecture for a film library

Figure 6.13 is an example of a system that is based on the client–server model. This is a multiuser, 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 include data about the film and video clips, and an e-commerce system that supports the sale of photographs, film, and video clips. The 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 cover distributed architectures in Chapter 17, where I explain the client–server model and its variants in more detail.

6.3.4 Pipe and filter architecture

My final example of a general Architectural pattern is the Pipe and Filter pattern (Figure 6.14). This is a model of the runtime 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.

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 6.15 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 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 architectural components that use incompatible data structures.

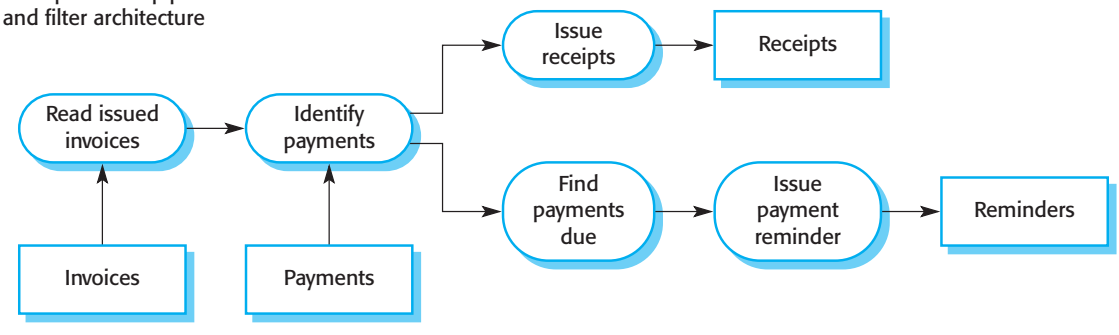
Figure 6.14 The Pipe and Filter pattern

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 such as billing systems. The architecture of an embedded system may also be organized as a process pipeline, with each process executing concurrently. I cover use of this pattern in embedded systems in Chapter 21.

An example of this type of system architecture, used in a batch processing application, is shown in Figure 6.15. An organization has issued invoices to customers. Once a week, payments that have been made are reconciled with the invoices. For

Figure 6.15 An example of the pipe and filter architecture



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

Pipe and filter systems are best suited to batch processing systems and embedded systems where there is limited user interaction. Interactive systems are difficult to write using the pipe and filter model because of the need for a stream of data to be processed. While 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 implement this as a sequential stream that conforms to the pipe and filter model.

6.4 Application architectures

Application systems are intended to meet a business or an 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 and meter calls, manage their network and issue bills to customers. 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 reimplemented when developing new systems. However, for many business systems, application architecture reuse is implicit when generic application systems are configured to create a new application. We see this in the widespread use of Enterprise Resource Planning (ERP) systems and off-the-shelf configurable application systems, such as systems for accounting and stock control. These systems have a standard architecture and components. The components are 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://software-engineering-book.com/web/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. You then specialize this for the specific system that is being developed.
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 applications* If you are discussing a specific application or trying to compare applications, then you can use the concepts identified in the generic architecture to talk about these applications.

There are many types of application system, and, in some cases, they may seem to be very different. However, superficially dissimilar applications may have much in common and thus share an abstract application architecture. I illustrate this by describing the 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 types 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 class of system includes interactive banking systems, e-commerce systems, information systems, and booking systems.

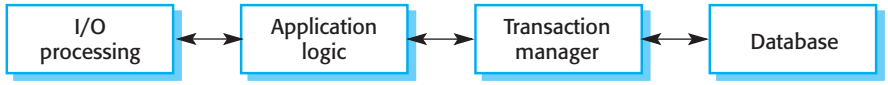


Figure 6.16 The structure of transaction processing applications

2. *Language processing systems* Language processing systems are systems in which the user's intentions are expressed in a formal language, such as a programming language. 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.

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 systems are designed to process user requests for information from a database, or requests to update a database (Lewis, Bernstein, and Kifer 2003). Technically, a database transaction is part of a sequence of operations and 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 a 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 database transaction is a customer request to withdraw money from a bank account using an ATM. This involves checking the customer account balance to see if sufficient funds are available, 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 6.16 illustrates the conceptual architectural structure of transaction processing 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 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

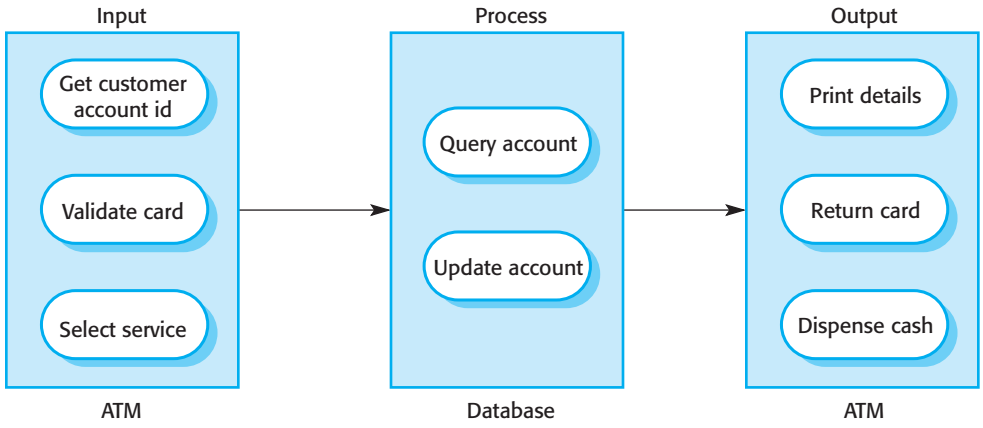


Figure 6.17 The software architecture of an ATM system

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 6.17 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. Information systems are almost always web-based systems, where the user interface is implemented in a web browser.

Figure 6.18 presents 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

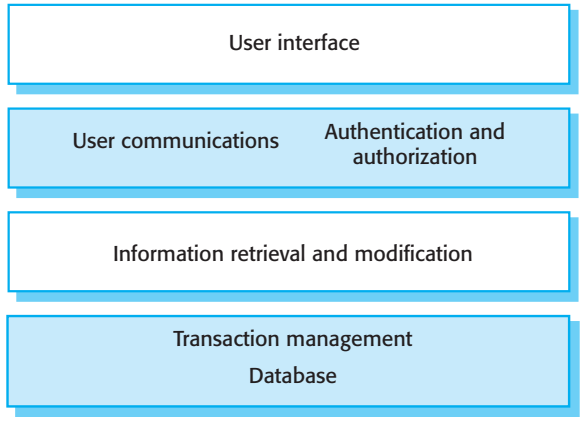


Figure 6.18 Layered information system architecture

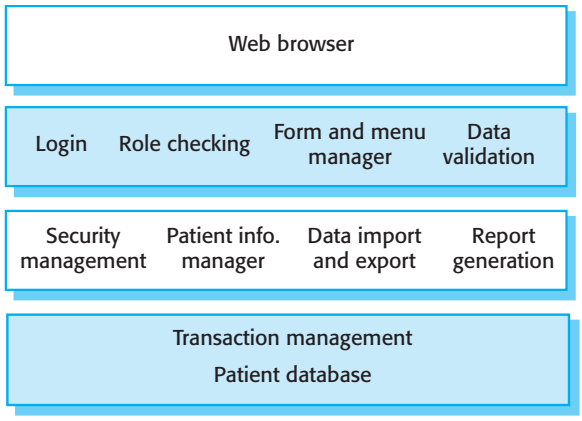


Figure 6.19 The architecture of the Mentcare system

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. The layers in this model can map directly onto servers in a distributed Internet-based system.

As an example of an instantiation of this layered model, Figure 6.19 shows the architecture of the Mentcare system. Recall that this system maintains and manages details of patients who are consulting specialist doctors about mental health problems. I have 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 a browser-based user interface.
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.
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 sometimes also transaction processing systems. 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

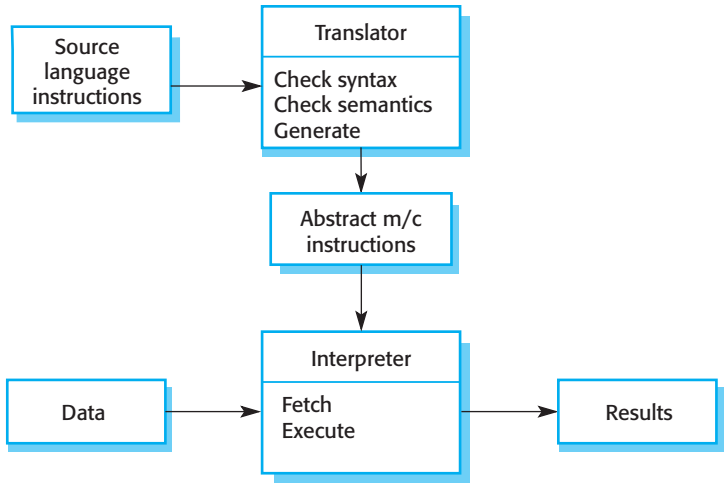


Figure 6.20 The architecture of a language processing system

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 6.18. These systems are often implemented as distributed systems with a multitier client server/architecture

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 thousands 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 one language into an alternative representation of that language and, for programming languages, may also execute the resulting code. Compilers translate a 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, for example, French to Norwegian.

A possible architecture for a language processing system for a programming language is illustrated in Figure 6.20. The source language instructions define the

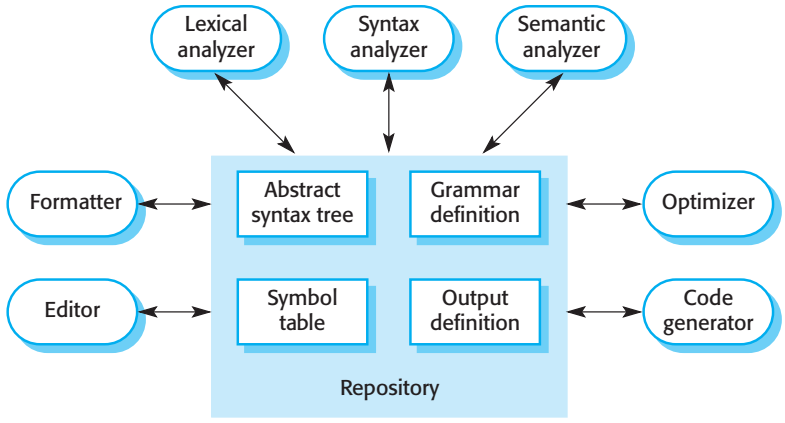


Figure 6.21 A repository architecture for a language processing system

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.

For many compilers, the interpreter is the system hardware that processes machine instructions, and the abstract machine is a real processor. However, for dynamically typed languages, such as Ruby or Python, the interpreter is a software component.

Programming language compilers that are part of a more general programming environment have a generic architecture (Figure 6.21) that includes the following components:

1. A lexical analyzer, which takes input language tokens and converts them into 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.
5. A semantic analyzer, which uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.
6. A code generator, which "walks" the syntax tree and generates abstract machine code.

Other components might also be included that analyze and transform the syntax tree to improve efficiency and remove redundancy from the generated machine code.

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

In other types of language processing system, such as a natural language translator, there will be additional components such as a dictionary. The output of the system is translation of the input text.

Figure 6.21 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. A program formatter can create listings of the program that highlight different syntactic elements and are therefore easier to read and understand.

Alternative Architectural patterns 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 6.22, 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 formatter. In this situation, changes from one component need to be reflected immediately in other components. It is better to organize the system around a repository, as shown in Figure 6.21 if you are implementing a general, language-oriented programming environment.

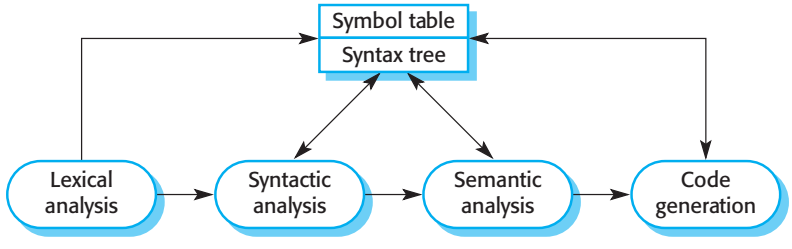


Figure 6.22 A pipe and filter compiler architecture