4

Software Architecture

The focus of this book is software products—individual applications that run on servers, personal computers, or mobile devices. To create a reliable, secure, and efficient product, you need to pay attention to its overall organization, how the software is decomposed into components, the server organization, and the technologies used to build the software. In short, you need to design the software architecture.

The architecture of a software product affects its performance, usability, security, reliability, and maintainability. Architectural issues are so important that three chapters in this book are devoted to them. In this chapter I discuss the decomposition of software into components, client–server architecture, and technology issues that affect the software architecture. In Chapter 5 I cover architectural issues and choices you have to make when implementing your software on the cloud. In Chapter 6 I cover microservices architecture, which is particularly important for cloud-based products.

If you google "software architecture definition," you find many different interpretations of the term. Some focus on "architecture" as a noun, the structure of a system; others consider "architecture" as a verb, the process of defining these structures. Rather than try to invent yet another definition, I use a definition of software architecture that is included in an IEEE standard,[1] shown in Table 4.1.

An important term in this definition is "components." Here it is used in a very general way, so a component can be anything from a program (large

[1]IEEE standard 1471. This has now been superseded by a later standard that has revised the definition. In my opinion, the revised definition is not an improvement and it is harder to explain and understand. https://en.wikipedia.org/wiki/IEEE_1471

Table 4.1 The IEEE definition of software architecture

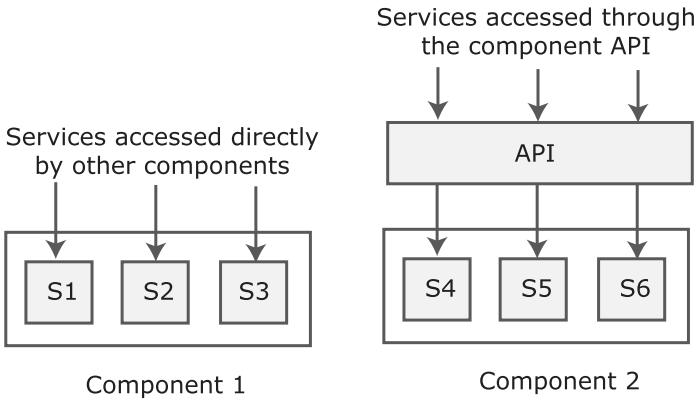
Software architecture
Architecture is the fundamental organization of a software system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.

scale) to an object (small scale). A component is an element that implements a coherent set of functionality or features. When designing software architecture, you don't have to decide how an architectural element or component is to be implemented. Rather, you design the component interface and leave the implementation of that interface to a later stage of the development process.

The best way to think of a software component is as a collection of one or more services that may be used by other components (Figure 4.1). A service is a coherent fragment of functionality. It can range from a large-scale service, such as a database service, to a microservice, which does one very specific thing. For example, a microservice might simply check the validity of a URL. Services can be implemented directly, as I discuss in Chapter 6, which covers microservice architecture. Alternatively, services can be part of modules or objects and accessed through a defined component interface or an application programming interface (API).

The initial enthusiasts for agile development invented slogans such as "You Ain't Gonna Need It" (YAGNI) and "Big Design Up Front" (BDUF), where YAGNI is good and BDUF is bad. They suggested that developers should not plan for change in their systems because change can't be predicted and might never happen. Many people think this means that agile developers believed there was no need to design the architecture of a software system

Figure 4.1 Access to services provided by software components



before implementation. Rather, when issues emerged during development, they should simply be tackled by refactoring—changing and reorganizing the software.

The inventors of agile methods are good engineers, and I don't think they intended that software architecture should not be designed. A principle of agile methods is that system planning and documentation should be minimized. However, this does not mean that you can't plan and describe the overall structure of your system. Agile methods now recognize the importance of architectural design and suggest that this should be an early activity in the development process. You can do this in a Scrum sprint where the outcome of the sprint is an informal architectural design.

Some people think it is best to have a single software architect. This person should be an experienced engineer who uses background knowledge and expertise to create a coherent architecture. However, the problems with this "single architect" model is that the team may not understand the architectural decisions that were made. To make sure that your whole team understands the architecture, I think everyone should be involved, in some way, in the architectural design process. This helps less experienced team members learn and understand why decisions are made. Furthermore, new team members may have knowledge and insights into new or unfamiliar technologies that can be used in the design and implementation of the software.

A development team should design and discuss the software product architecture before starting the final product implementation. They should agree on priorities and understand the trade-offs that they are making in these architectural decisions. They should create a description of the product architecture that sets out the fundamental structure of the software and serves as a reference for its implementation.

4.1 Why is architecture important?

I suggested in Chapter 1 that you should always develop a product prototype. The aim of a prototype is to help you understand more about the product that you are developing, and so you should aim to develop this as quickly as possible. Issues such as security, usability, and long-term maintainability are not important at this stage.

When you are developing a final product, however, "non-functional" attributes are critically important (Table 4.2). It is these attributes, rather than

Table 4.2 Non-functional system quality attributes

Attribute	Key issue
Responsiveness	Does the system return results to users in a reasonable time?
Reliability	Do the system features behave as expected by both developers and users?
Availability	Can the system deliver its services when requested by users?
Security	Does the system protect itself and users' data from unauthorized attacks and intrusions?
Usability	Can system users access the features that they need and use them quickly and without errors?
Maintainability	Can the system be readily updated and new features added without undue costs?
Resilience	Can the system continue to deliver user services in the event of partial failure or external attack?

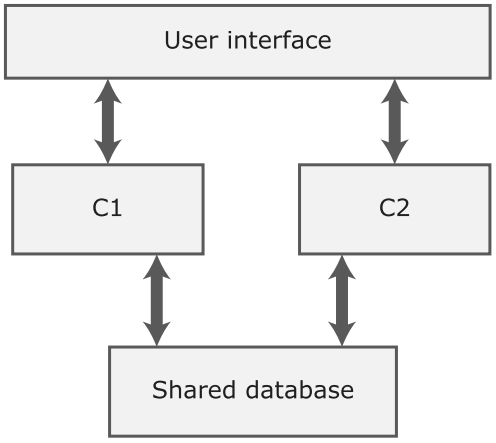
product features, that influence user judgements about the quality of your software. If your product is unreliable, insecure, or difficult to use, then it is almost bound to be a failure. Product development takes much longer than prototyping because of the time and effort that are needed to ensure that your product is reliable, maintainable, secure, and so on.

Architecture is important because the architecture of a system has a fundamental influence on these non-functional properties. Table 4.3 is a

Table 4.3 The influence of architecture on system security

A centralized security architecture
In the *Star Wars* prequel *Rogue One* (https://en.wikipedia.org/wiki/Rogue_One), the evil Empire has stored the plans for all of their equipment in a single, highly secure, well-guarded, remote location. This is called a centralized security architecture. It is based on the principle that if you maintain all of your information in one place, then you can apply lots of resources to protect that information and ensure that intruders can't get it. Unfortunately (for the Empire), the rebels managed to breach their security. They stole the plans for the Death Star, an event that underpins the whole *Star Wars* saga. In trying to stop them, the Empire destroyed their entire archive of system documentation with who knows what resultant costs. Had the Empire chosen a distributed security architecture, with different parts of the Death Star plans stored in different locations, then stealing the plans would have been more difficult. The rebels would have had to breach security in all locations to steal the complete Death Star blueprints.

Figure 4.2 Shared database architecture



non-computing example of how architectural choices affect system properties. It is taken from the film *Rogue One*, part of the *Star Wars* saga.

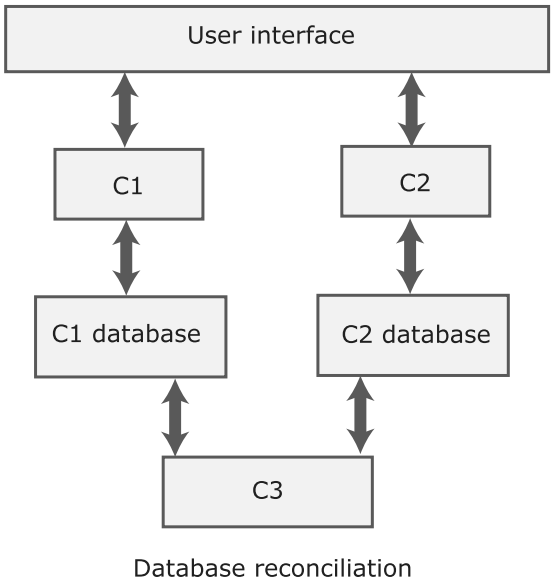
Rogue One is science fiction, but it demonstrates that architectural decisions have fundamental consequences. The benefits of a centralized security architecture are that it is easy to design and build protection and the protected information can be accessed efficiently. However, if your security is breached, you lose everything. If you distribute information, it takes longer to access all of the information and costs more to protect it. If security is breached in one location, however, you lose only the information that you have stored there.

Figures 4.2 and 4.3 illustrate a situation where the system architecture affects the maintainability and performance of a system. Figure 4.2 shows a system with two components (C1 and C2) that share a common database. This is a common architecture for web-based systems. Let's assume that C1 runs slowly because it has to reorganize the information in the database before using it. The only way to make C1 faster might be to change the database. This means that C2 also has to be changed, which may potentially affect its response time.

Figure 4.3 shows a different architecture where each component has its own copy of the parts of the database that it needs. Each of these components can therefore use a different database structure, and so operate efficiently. If one component needs to change the database organization, this does not affect the other component. Also, the system can continue to provide a partial service in the event of a database failure. This is impossible in a centralized database architecture.

However, the distributed database architecture may run more slowly and may cost more to implement and change. There needs to be a mechanism

Figure 4.3 Multiple database architecture



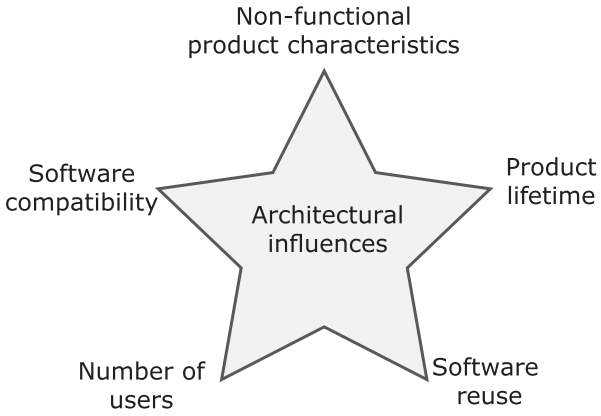
(shown here as component C3) to ensure that the data shared by C1 and C2 are kept consistent when changes are made. This takes time and it is possible that users will occasionally see inconsistent information. Furthermore, additional storage costs are associated with the distributed database architecture and higher costs of change if a new component that requires its own database has to be added to the system.

It is impossible to optimize all of the non-functional attributes in the same system. Optimizing one attribute, such as security, inevitably affects other attributes, such as system usability and efficiency. You have to think about these issues and the software architecture before you start programming. Otherwise, it is almost inevitable that your product will have undesirable characteristics and will be difficult to change.

Another reason why architecture is important is that the software architecture you choose affects the complexity of your product. The more complex a system, the more difficult and expensive it is to understand and change. Programmers are more likely to make mistakes and introduce bugs and security vulnerabilities when they are modifying or extending a complex system. Therefore, minimizing complexity should be an important goal for architectural design.

The organization of a system has a profound effect on its complexity, and it is very important to pay attention to this when designing the software architecture. I explain architectural complexity in Section 4.3, and I cover general issues of program complexity in Chapter 8.

Figure 4.4 Issues that influence architectural decisions



4.2 Architectural design

Architectural design involves understanding the issues that affect the architecture of your particular product and creating a description of the architecture that shows the critical components and some of their relationships. The architectural issues that are most important for software product development are shown in Figure 4.4 and Table 4.4.

Other human and organizational factors also affect architectural design decisions. These include the planned schedule to bring the product to market, the capabilities of your development team, and the software development budget. If you choose an architecture that requires your team to learn unfamiliar technologies, then this may delay the delivery of your system. There is no point in creating a "perfect" architecture that is delivered late if this means that a competing product captures the market.

Architectural design involves considering these issues and deciding on essential compromises that allow you to create a system that is "good enough" and can be delivered on time and on budget. Because it is impossible to optimize everything, you have to make a series of trade-offs when choosing an architecture for your system. Some examples are:

- maintainability vs. performance;
- security vs. usability;
- availability vs. time to market and cost.

Table 4.4 The importance of architectural design issues

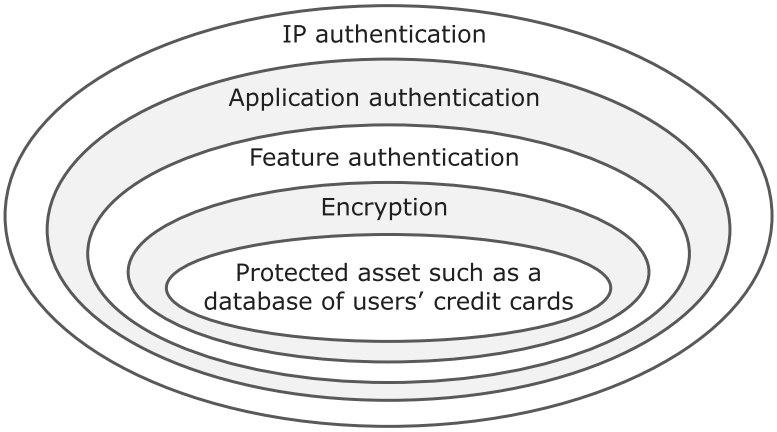
Issue	Architectural importance
Non-functional product characteristics	Non-functional product characteristics such as security and performance affect all users. If you get these wrong, your product is unlikely to be a commercial success. Unfortunately, some characteristics are opposing, so you can optimize only the most important.
Product lifetime	If you anticipate a long product lifetime, you need to create regular product revisions. You therefore need an architecture that can evolve, so that it can be adapted to accommodate new features and technology.
Software reuse	You can save a lot of time and effort if you can reuse large components from other products or open-source software. However, this constrains your architectural choices because you must fit your design around the software that is being reused.
Number of users	If you are developing consumer software delivered over the Internet, the number of users can change very quickly. This can lead to serious performance degradation unless you design your architecture so that your system can be quickly scaled up and down.
Software compatibility	For some products, it is important to maintain compatibility with other software so that users can adopt your product and use data prepared using a different system. This may limit architectural choices, such as the database software that you can use.

System maintainability is an attribute that reflects how difficult and expensive it is to make changes to a system after it has been released to customers. In general, you improve maintainability by building a system from small self-contained parts, each of which can be replaced or enhanced if changes are required. Wherever possible, you should avoid shared data structures and you should make sure that, when data are processed, separate components are used to "produce" and to "consume" data.

In architectural terms, this means that the system should be decomposed into fine-grain components, each of which does one thing and one thing only. More general functionality emerges by creating networks of these components that communicate and exchange information. Microservice architectures, explained in Chapter 6, are an example of this type of architecture.

However, it takes time for components to communicate with each other. Consequently, if many components are involved in implementing a product feature,

Figure 4.5 Authentication layers



the software will be slower. Avoiding shared data structures also has an impact on performance. There may be delays involved in transferring data from one component to another and in ensuring that duplicated data are kept consistent.

The constant and increasing risk of cybercrime means that all product developers have to design security into their software. Security is so important for product development that I devote a separate chapter (Chapter 7) to this topic. You can achieve security by designing the system protection as a series of layers (Figure 4.5). An attacker has to penetrate all of those layers before the system is compromised. Layers might include system authentication layers, a separate critical feature authentication layer, an encryption layer, and so on. Architecturally, you can implement each of these layers as separate components so that if an attacker compromises one of these components, then the other layers remain intact.

Unfortunately, there are drawbacks to using multiple authentication layers. A layered approach to security affects the usability of the software. Users have to remember information, like passwords, that is needed to penetrate a security layer. Their interaction with the system is inevitably slowed by its security features. Many users find this irritating and often look for work-arounds so that they do not have to re-authenticate to access system features or data.

Many security breaches arise because users behave in an insecure way, such as choosing passwords that are easy to guess, sharing passwords, and leaving systems logged on. They do this because they are frustrated by system security features that are difficult to use or that slow down their access to the system and its data. To avoid this, you need an architecture that doesn't have too many security layers, that doesn't enforce unnecessary security, and that provides, where possible, helper components that reduce the load on users.

The availability of a system is a measure of the amount of uptime of that system. It is normally expressed as a percentage of the time that a system is available to deliver user services. Therefore, an availability of 99.9% in a system that is intended to be constantly available means that the system should be available for 86,313 seconds out of the 86,400 seconds in a day. Availability is particularly important in enterprise products, such as products for the finance industry, where 24/7 operation is expected.

Architecturally, you improve availability by having redundant components in a system. To make use of redundancy, you include sensor components that detect failure and switching components that switch operation to a redundant component when a failure is detected. The problem here is that implementing these extra components takes time and increases the cost of system development. It adds complexity to the system and therefore increases the chances of introducing bugs and vulnerabilities. For this reason, most product software does not use component-switching in the event of system failure. As I explain in Chapter 8, you can use reliable programming techniques to reduce the changes of system failure.

Once you have decided on the most important quality attributes for your software, you have to consider three questions about the architectural design of your product:

1. How should the system be organized as a set of architectural components, where each of these components provides a subset of the overall system functionality? The organization should deliver the system security, reliability, and performance that you need.
2. How should these architectural components be distributed and communicate with each other?
3. What technologies should be used in building the system, and what components should be reused?

I cover these three questions in the remaining sections of this chapter.

Architectural descriptions in product development provide a basis for the development team to discuss the organization of the system. An important secondary role is to document a shared understanding of what needs to be developed and what assumptions have been made in designing the software. The final system may differ from the original architectural model, so it is not a reliable way of documenting delivered software.

I think informal diagrams based around icons to represent entities, lines to represent relationships, and text are the best way to describe and share

information about software product architectures. Everyone can participate in the design process. You can draw and change informal diagrams quickly without using special software tools. Informal notations are flexible so that you can make unanticipated changes easily. New people joining a team can understand them without specialist knowledge.

The main problems with informal models are that they are ambiguous and they can't be checked automatically for omissions and inconsistencies. If you use more formal approaches, based on architectural description languages (ADLs) or the Unified Modeling Language (UML), you reduce ambiguity and can use checking tools. However, my experience is that formal notations get in the way of the creative design process. They constrain expressiveness and require everyone to understand them before they can participate in the design process.

4.3 System decomposition

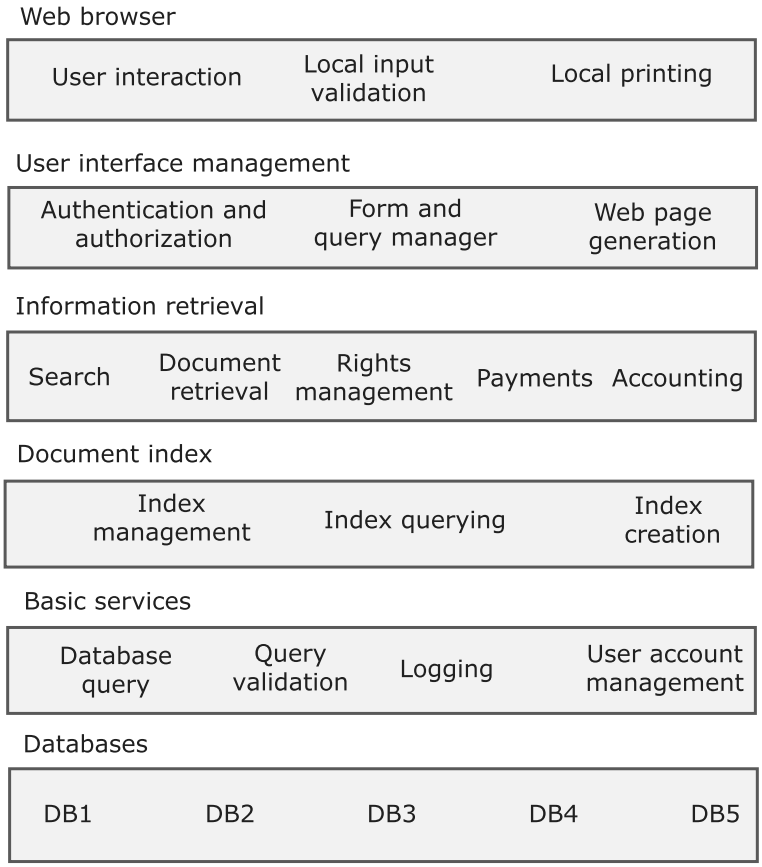
The idea of abstraction is fundamental to all software design. Abstraction in software design means that you focus on the essential elements of a system or software component without concern for its details. At the architectural level, your concern should be on large-scale architectural components. Decomposition involves analyzing these large-scale components and representing them as a set of finer-grain components.

For example, Figure 4.6 is a diagram of the architecture of a product that I was involved with some years ago. This system was designed for use in libraries and gave users access to documents that were stored in a number of private databases, such as legal and patent databases. Payment was required for access to these documents. The system had to manage the rights to these documents and collect and account for access payments.

In this diagram, each layer in the system includes a number of logically related components. Informal layered models, like Figure 4.6, are widely used to show how a system is decomposed into components, with each component providing significant system functionality.

Web-based and mobile systems are event-based systems. An event in the user interface, such as a mouse click, triggers the actions to implement the user's choice. This means that the flow of control in a layered system is top-down. User events in the higher layers trigger actions in that layer that, in turn, trigger events in lower layers. By contrast, most information flows in the

Figure 4.6 An architectural model of a document retrieval system



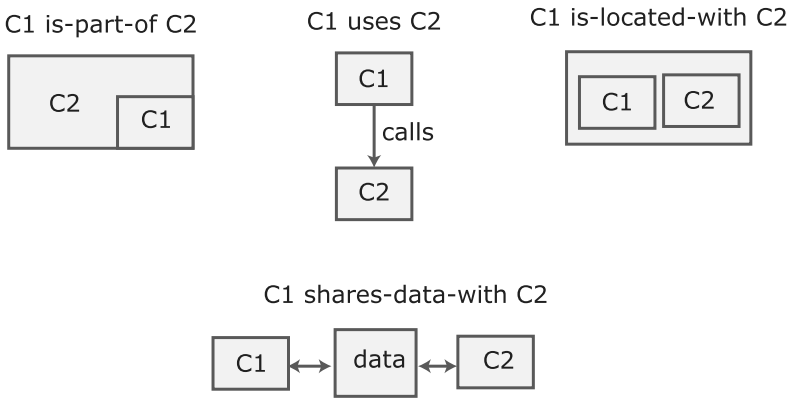
system are bottom-up. Information is created at lower layers, is transformed in the intermediate layers, and is finally delivered to users at the top level.

There is often confusion about architectural terminology, words such as "service," "component," and "module." There are no standard, widely accepted definitions of these terms, but I try to use them consistently in this chapter and elsewhere in the book:

1. A *service* is a coherent unit of functionality. This may mean different things at different levels of the system. For example, a system may offer an email service and this email service may itself include services for creating, sending, reading, and storing email.

2. A *component* is a named software unit that offers one or more services to other software components or to the end-users of the software. When used by other components, these services are accessed through an API. Components may use several other components to implement their services.

Figure 4.7 Examples of component relationships



3. A *module* is a named set of components. The components in a module should have something in common. For example, they may provide a set of related services.

Complexity in a system architecture arises because of the number and the nature of the relationships among components in that system. I discuss this in more detail in Chapter 8. When you change a program, you have to understand these relationships to know how changes to one component affect other components. When decomposing a system into components, you should try to avoid introducing unnecessary complexity into the software.

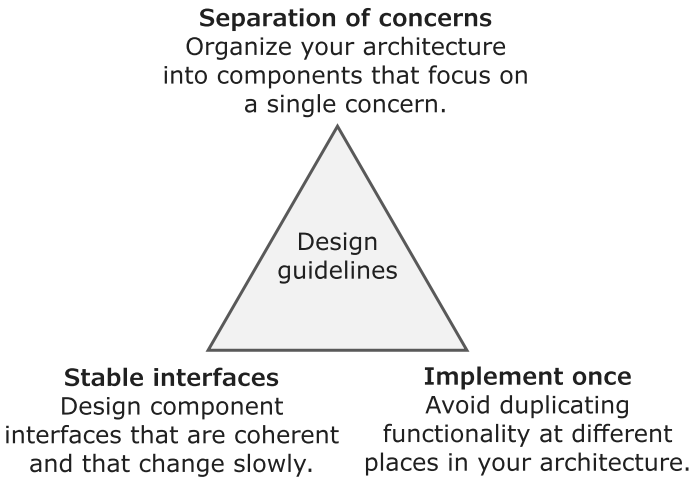
Components have different types of relationships with other components (Figure 4.7). Because of these relationships, when you make a change to one component, you often need to make changes to several other components.

Figure 4.7 shows four types of component relationship:

1. *Part-of* One component is part of another component. For example, a function or method may be part of an object.
2. *Uses* One component uses the functionality provided by another component.
3. *Is-located-with* One component is defined in the same module or object as another component.
4. *Shares-data-with* A component shares data with another component.

As the number of components increases, the number of relationships tends to increase at a faster rate. This is the reason large systems are more complex than small systems. It is impossible to avoid complexity increasing with the

Figure 4.8 Architectural design guidelines



size of the software. However, you can control architectural complexity by doing two things:

1. *Localize relationships* If there are relationships between components A and B (say), they are easier to understand if A and B are defined in the same module. You should identify logical component groupings (such as the layers in a layered architecture) with relationships mostly within a component group.
2. *Reduce shared dependencies* Where components A and B depend on some other component or data, complexity increases because changes to the shared component mean you have to understand how these changes affect both A and B. It is always preferable to use local data wherever possible and to avoid sharing data if you can.

Three general design guidelines help to control complexity, as shown in Figure 4.8.

The *separation of concerns* guideline suggests that you should identify relevant architectural concerns in groupings of related functionality. Examples of architectural concerns are user interaction, authentication, system monitoring, and database management. Ideally, you should be able to identify the components or groupings of components in your architecture that are related to each concern. At a lower level, separation of concerns means that components should, ideally, do only one thing. I cover separation of concerns in more detail in Chapter 8.

The *implement once* guideline suggests that you should not duplicate functionality in your software architecture. This is important, as duplication can cause problems when changes are made. If you find that more than one architectural component needs or provides the same or a similar service, you should reorganize your architecture to avoid duplication.

You should never design and implement software where components know of and rely on the implementation of other components. Implementation dependencies mean that if a component is changed, then the components that rely on its implementation also have to be changed. Implementation details should be hidden behind a component interface (API).

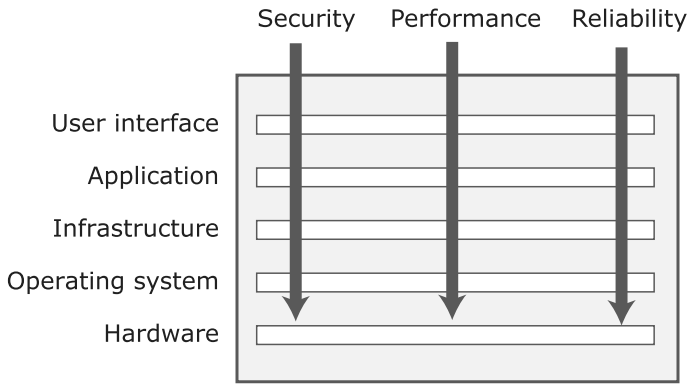
The *stable interfaces* guideline is important so that components that use an interface do not have to be changed because the interface has changed.

Layered architectures, such as the document retrieval system architecture shown in Figure 4.6, are based on these general design guidelines:

1. Each layer is an area of concern and is considered separately from other layers. The top layer is concerned with user interaction, the next layer down with user interface management, the third layer with information retrieval, and so on.
2. Within each layer, the components are independent and do not overlap in functionality. The lower layers include components that provide general functionality, so there is no need to replicate this in the components in a higher level.
3. The architectural model is a high-level model that does not include implementation information. Ideally, components at level X (say) should only interact with the APIs of the components in level X-1; that is, interactions should be between layers and not across layers. In practice, however, this is often impossible without code duplication. The lower levels of the stack of layers provide basic services that may be required by components that are not in the immediate level above them. It makes no sense to add additional components in a higher layer if these are used only to access lower-level components.

Layered models are informal and are easy to draw and understand. They can be drawn on a whiteboard so that the whole team can see how the system is decomposed. In a layered model, components in lower layers should never depend on higher-level components. All dependencies should be on lower-level components. This means that if you change a component at level X in the stack, you should not have to make changes to components at lower levels in the stack. You only have to consider the effects of the change on components at higher levels.

Figure 4.9 Cross-cutting concerns



The layers in the architectural model are not components or modules but are simply logical groupings of components. They are relevant when you are designing the system, but you can't normally identify these layers in the system implementation.

The general idea of controlling complexity by localizing concerns within a single layer of an architecture is a compelling one. If you can do this, you don't have to change components in other layers when components in any one layer are modified. Unfortunately, there are two reasons why localizing concerns may not always be possible:

1. For practical reasons of usability and efficiency, it may be necessary to divide functionality so that it is implemented in different layers.
2. Some concerns are cross-cutting concerns and have to be considered at every layer in the stack.

You can see an example of the problem of practical separation of concerns in Figure 4.6. The top layer includes "Local input validation" and the fifth layer in the stack includes "Query validation." The "validation concern" is not implemented in a single lower-level server component because this is likely to generate too much network traffic.

If user data validation is a server rather than a browser operation, this requires a network transaction for every field in a form. Obviously, this slows the system down. Therefore, it makes sense to implement some local input checking, such as date checking, in the user's browser or mobile app. Some checking, however, may require knowledge of database structure or a user's permissions, and this can be carried out only when all of the form has been completed. As I explain in Chapter 7, the checking of security-critical fields should also be a server-side operation.

Cross-cutting concerns are systemic concerns; that is, they affect the whole system. In a layered architecture, cross-cutting concerns affect all layers in the system as well as the way in which people use the system. Figure 4.9 shows

Table 4.5 Security as a cross-cutting concern

Security architecture
Different technologies are used in different layers, such as an SQL database or a Firefox browser. Attackers can try to use vulnerabilities in these technologies to gain access. Consequently, you need protection from attacks at each layer as well as protection at lower layers in the system from successful attacks that have occurred at higher-level layers. If there is only a single security component in a system, this represents a critical system vulnerability. If all security checking goes through that component and it stops working properly or is compromised in an attack, then you have no reliable security in your system. By distributing security across the layers, your system is more resilient to attacks and software failure (remember the *Rogue One* example earlier in the chapter).

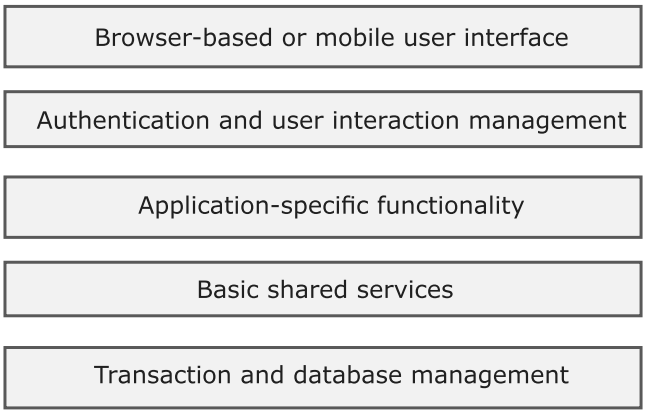
three cross-cutting concerns—security, performance and reliability—that are important for software products.

Cross-cutting concerns are completely different from the functional concerns represented by layers in a software architecture. Every layer has to take them into account, and there are inevitably interactions between the layers because of these concerns. These cross-cutting concerns make it difficult to improve system security after it has been designed. Table 4.5 explains why security cannot be localized in a single component or layer.

Let's assume that you are a software architect and you want to organize your system into a series of layers to help control complexity. You are then faced with the general question "Where do I start?". Fortunately, many software products that are delivered over the web have a common layered structure that you can use as a starting point for your design. This common structure is shown in Figure 4.10. The functionality of the layers in this generic layered architecture is explained in Table 4.6.

For web-based applications, the layers shown in Figure 4.10 can be the starting point for your decomposition process. The first stage is to think about whether this five-layer model is the right one or whether you need more or fewer layers. Your aim should be for layers to be logically coherent, so that all components in a layer have something in common. This may mean that you need one or more additional layers for your application-specific functionality. Sometimes you may wish to have authentication in a separate layer, and sometimes it makes sense to integrate shared services with the database management layer.

Figure 4.10 A generic layered architecture for a web-based application



Once you have figured out how many layers to include in your system, you can start to populate these layers. In my experience, the best way to do this is to involve the whole team and try out various decompositions to help understand their advantages and disadvantages. This is a trial-and-error process; you stop when you have what seems to be a workable decomposition architecture.

Table 4.6 Layer functionality in a web-based application

Layer	Explanation
Browser-based or mobile user interface	A web browser system interface in which HTML forms are often used to collect user input. Javascript components for local actions, such as input validation, should also be included at this level. Alternatively, a mobile interface may be implemented as an app.
Authentication and UI management	A user interface management layer that may include components for user authentication and web page generation.
Application-specific functionality	An “application” layer that provides functionality of the application. Sometimes this may be expanded into more than one layer.
Basic shared services	A shared services layer that includes components that provide services used by the application layer components.
Database and transaction management	A database layer that provides services such as transaction management and recovery. If your application does not use a database, then this may not be required.

Table 4.7 iLearn architectural design principles

Principle	Explanation
Replaceability	It should be possible for users to replace applications in the system with alternatives and to add new applications. Consequently, the list of applications included should not be hard-wired into the system.
Extensibility	It should be possible for users or system administrators to create their own versions of the system, which may extend or limit the "standard" system.
Age-appropriate	Alternative user interfaces should be supported so that age-appropriate interfaces for students at different levels can be created.
Programmability	It should be easy for users to create their own applications by linking existing applications in the system.
Minimum work	Users who do not wish to change the system should not have to do extra work so that other users can make changes.

The discussion about system decomposition may be driven by fundamental principles that should apply to the design of your application system. These set out goals that you wish to achieve. You can then evaluate architectural design decisions against these goals. For example, Table 4.7 shows the principles that we thought were most important when designing the iLearn system architecture.

Our goal in designing the iLearn system was to create an adaptable, universal system that could be updated easily as new learning tools became available. This means it must be possible to change and replace components and services in the system (principles 1 and 2). Because the potential system users ranged in age from 3 to 18, we needed to provide age-appropriate user interfaces and make it easy to choose an interface (principle 3). Principle 4 also contributes to system adaptability, and principle 5 was included to ensure that this adaptability did not adversely affect users who did not require it.

Unfortunately, principle 1 may sometimes conflict with principle 4. If you allow users to create new functionality by combining applications, then these combined applications may not work if one or more of the constituent applications are replaced. You often have to address this kind of conflict in architectural design.

These principles led us to an architectural design decision that the iLearn system should be service-oriented. Every component in the system is a service. Any service is potentially replaceable, and new services can be created

by combining existing services. Different services that deliver comparable functionality can be provided for students of different ages.

Using services means that the potential conflict I identified above is mostly avoidable. If a new service is created by using an existing service and, subsequently, other users want to introduce an alternative, they may do so. The older service can be retained in the system, so that users of that service don't have to do more work because a newer service has been introduced.

We assumed that only a minority of users would be interested in programming their own system versions. Therefore, we decided to provide a standard set of application services that had some degree of integration with other services. We anticipated that most users would rely on these and would not wish to replace them. Integrated application services, such as blogging and wiki services, could be designed to share information and make use of common shared services. Some users may wish to introduce other services into their environment, so we also allowed for services that were not tightly integrated with other system services.

We decided to support three types of application service integration:

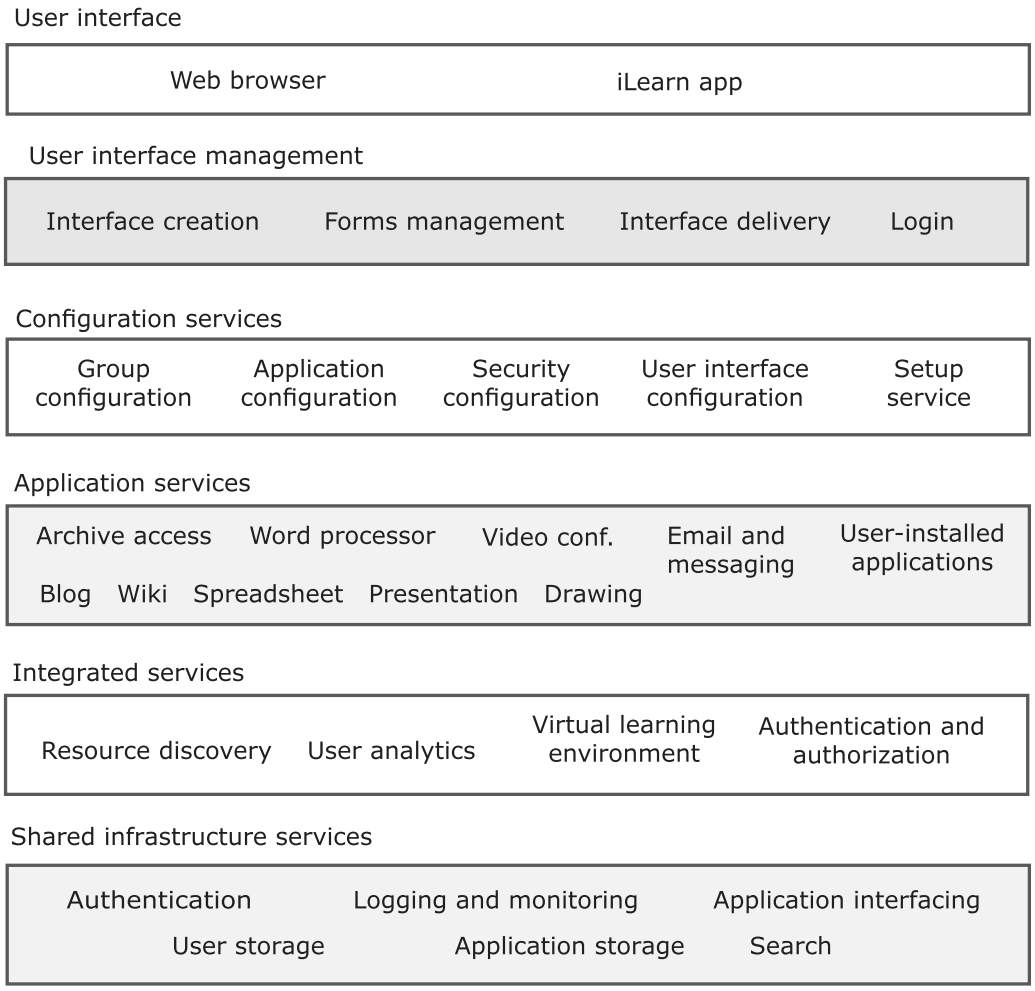
1. *Full integration* Services are aware of and can communicate with other services through their APIs. Services may share system services and one or more databases. An example of a fully integrated service is a specially written authentication service that checks the credentials of system users.

2. *Partial integration* Services may share service components and databases, but they are not aware of and cannot communicate directly with other application services. An example of a partially integrated service is a Wordpress service in which the Wordpress system was changed to use the standard authentication and storage services in the system. Office 365, which can be integrated with local authentication systems, is another example of a partially integrated service that we included in the iLearn system.

3. *Independent* These services do not use any shared system services or databases, and they are unaware of any other services in the system. They can be replaced by any other comparable service. An example of an independent service is a photo management system that maintains its own data.

The layered model for the iLearn system that we designed is shown in Figure 4.11. To support application "replaceability", we did not base the system around a shared database. However, we assumed that fully-integrated applications would use shared services such as storage and authentication.

Figure 4.11 A layered architectural model of the iLearn system



To support the requirement that users should be able to configure their own version of an iLearn system, we introduced an additional layer into the system, above the application layer. This layer includes several components that incorporate knowledge of the installed applications and provide configuration functionality to end-users.

The system has a set of pre-installed application services. Additional application services can be added or existing services replaced by using the application configuration facilities. Most of these application services are independent and manage their own data. Some services are partially integrated, however, which simplifies information sharing and allows more detailed user information to be collected.

The fully integrated services have to be specially written or adapted from open-source software. They require knowledge of how the system is used

and access to user data in the storage system. They may make use of other services at the same level. For example, the user analytics service provides information about how individual students use the system and can highlight problems to teachers. It needs to be able to access both log information and student records from the virtual learning environment.

System decomposition has to be done in conjunction with choosing technologies for your system (see Section 4.5). The reason for this is that the choice of technology used in a particular layer affects the components in the layers above. For example, you may decide to use a relational database technology as the lowest layer in your system. This makes sense if you are dealing with well-structured data. However, your decision affects the components to be included in the services layer because you need to be able to communicate with the database. You may have to include components to adapt the data passed to and from the database.

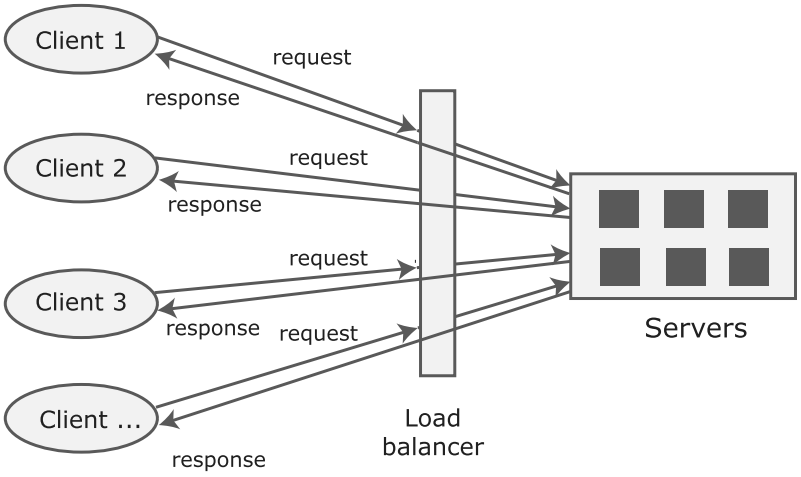
Another important technology-related decision is the interface technologies that you will use. This choice depends on whether you will be supporting browser interfaces only (often the case with business systems) or you also want to provide interfaces on mobile devices. If you are supporting mobile devices, you need to include components to interface with the relevant iOS and Android UI development toolkits.

4.4 Distribution architecture

The majority of software products are now web-based products, so they have a client–server architecture. In this architecture, the user interface is implemented on the user's own computer or mobile device. Functionality is distributed between the client and one or more server computers. During the architectural design process, you have to decide on the "distribution architecture" of the system. This defines the servers in the system and the allocation of components to these servers.

Client–server architectures are a type of distribution architecture that is suited to applications in which clients access a shared database and business logic operations on those data. Figure 4.12 shows a logical view of a client–server architecture that is widely used in web-based and mobile software products. These applications include several servers, such as web servers and database servers. Access to the server set is usually mediated by a load balancer, which distributes requests to servers. It is designed to ensure that the computing load is evenly shared by the set of servers.

Figure 4.12 Client–server architecture



The client is responsible for user interaction, based on data passed to and from the server. When this architecture was originally devised, clients were character terminals with hardly any local processing capability. The server was a mainframe computer. All processing was carried out on the server, with the client handling only user interaction. Now clients are computers or mobile devices with lots of processing power, so most applications are designed to include significant client-side processing.

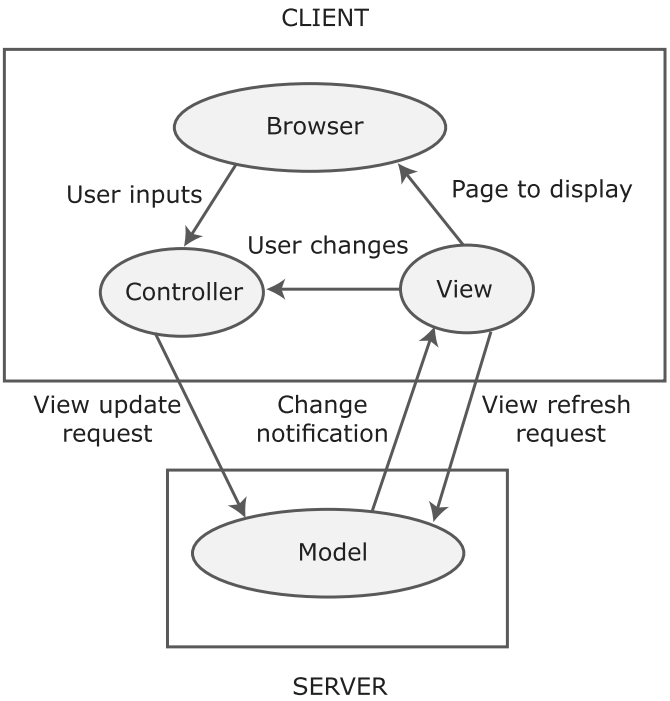
Client–server interaction is usually organized using what is called the Model-View-Controller (MVC) pattern. This architectural pattern is used so that client interfaces can be updated when data on the server change (Figure 4.13).

The term "model" is used to mean the system data and the associated business logic. The model is always shared and maintained on the server. Each client has its own view of the data, with views responsible for HTML page generation and forms management. There may be several views on each client in which the data are presented in different ways. Each view registers with the model so that when the model changes, all views are updated. Changing the information in one view leads to all other views of the same information being updated.

User inputs that change the model are handled by the controller. The controller sends update requests to the model on the server. It may also be responsible for local input processing, such as data validation.

The MVC pattern has many variants. In some, all communication between the view and the model goes through the controller. In others, views can also handle user inputs. However, the essence of all of these variants is that the model is decoupled from its presentation. It can, therefore, be presented in different ways and each presentation can be independently updated when changes to the data are made.

Figure 4.13 The Model-View-Controller pattern



For web-based products, Javascript is mostly used for client-side programming. Mobile apps are mostly developed in Java (Android) and Swift (iOS). I don't have experience in mobile app development, so I focus here on web-based products. However, the underlying principles of interaction are the same.

Client–server communication normally uses the HTTP protocol, which is a text-based request/response protocol. The client sends a message to the server that includes an instruction such as GET or POST along with the identifier of a resource (usually a URL) on which that instruction should operate. The message may also include additional information, such as information collected from a form. So, a database update may be encoded as a POST instruction, an identifier for the information to be updated plus the changed information input by the user. Servers do not send requests to clients, and clients always wait for a response from the server.[2]

HTTP is a text-only protocol, so structured data must be represented as text. Two ways of representing these data are widely used—namely, XML and JSON. XML is a markup language with tags used to identify each data item. JSON is

[2]This is not strictly true if a technology such as Node.js is used to build server-side applications. This allows both clients and servers to generate requests and responses. However, the general client–server model still applies.

Program 4.1 An example of JSON information representation

```
{
"book": [
{
“title”:   ”Software Engineering”,
"author": “Ian Sommerville”,
“publisher”: “Pearson Higher Education”,
“place”: “Hoboken, NJ”,
 “year”: “2015”,
"edition": “10th”,
 “ISBN”: “978-0-13-394303-0”
},
 ]
}
```

a simpler representation based on the representation of objects in the Javascript language. Usually JSON representations are more compact and faster to parse than XML text. I recommend that you use JSON for data representation.

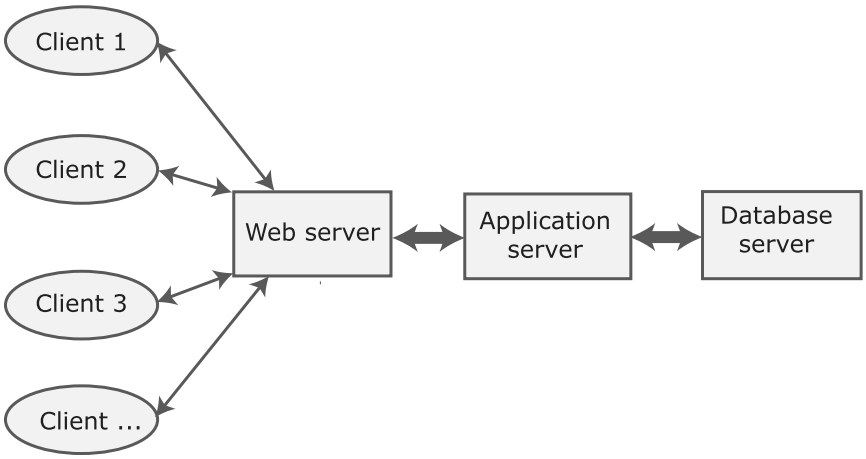
As well as being faster to process than XML, JSON is easier for people to read. Program 4.1 shows the JSON representation of cataloging information about a software engineering textbook.

There are good JSON tutorials available on the web, so I don’t go into more detail about this notation.

Many web-based applications use a multi-tier architecture with several communicating servers, each with its own responsibilities. Figure 4.14 illustrates the distributed components in a multi-tier web-based system architecture. For simplicity, I assume there is only a single instance of each of these servers and so a load balancer is not required. Web-based applications usually include three types of server:

1. A web server that communicates with clients using the HTTP protocol. It delivers web pages to the browser for rendering and processes HTTP requests from the client.
2. An application server that is responsible for application-specific operations. For example, in a booking system for a theater, the application server provides information about the shows as well as basic functionality that allows a theatergoer to book seats for shows.
3. A database server that manages the system data and transfers these data to and from the system database.

Figure 4.14 Multi-tier client–server architecture



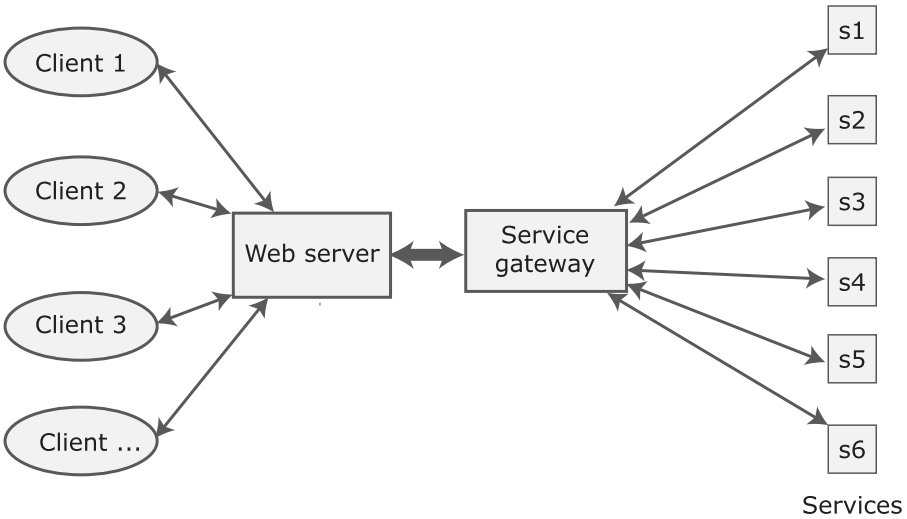
Sometimes a multi-tier architecture may use additional specialized servers. For example, in a theater booking system, the user's payments may be handled by a credit card payment server provided by a company that specializes in credit card payments. This makes sense for most e-commerce applications, as a high cost is involved in developing a trusted payment system. Another type of specialized server that is commonly used is an authentication server. This checks users' credentials when they log in to the system.

An alternative to a multi-tier client–server architecture is a service-oriented architecture (Figure 4.15) where many servers may be involved in providing services. Services in a service-oriented architecture are stateless components, which means that they can be replicated and can migrate from one computer to another. A service-oriented architecture is usually easier to scale as demand increases and is resilient to failure.

The services shown in Figure 4.15 are services that support features in the system. These are the services provided by the application layer and layers above this in the decomposition stack. To keep the diagram simple, I do not show interactions between services or infrastructure services that provide functionality from lower levels in the decomposition. Service-oriented architectures are increasingly used, and I discuss them in more detail in Chapter 6.

We chose a service-oriented distribution architecture for the iLearn system, with each of the components shown in Figure 4.11 implemented as a separate service. We chose this architecture because we wanted to make it easy to update the system with new functionality. It also simplified the problem of adding new, unforeseen services to the system.

Figure 4.15 Service-oriented architecture



Multi-tier and service-oriented architectures are the main types of distribution architecture for web-based and mobile systems. You have to decide which of these to choose for your software product. The issues that you must consider are:

1. *Data type and data updates* If you are mostly using structured data that may be updated by different system features, it is usually best to have a single shared database that provides locking and transaction management. If data are distributed across services, you need a way to keep them consistent, and this adds overhead to your system.
2. *Frequency of change* If you anticipate that system components will regularly be changed or replaced, then isolating these components as separate services simplifies those changes.
3. *The system execution platform* If you plan to run your system on the cloud with users accessing it over the Internet, it is usually best to implement it as a service-oriented architecture because scaling the system is simpler. However, if your product is a business system that runs on local servers, a multi-tier architecture may be more appropriate.

When I wrote this book in 2018, the distribution architecture of most business software products was a multi-tier client–server architecture with user interaction implemented using the MVC pattern. However, these products are increasingly being updated to use service-oriented architectures, running on public cloud platforms. I think that, over time, this type of architecture will become the norm for web-based software products.

Table 4.8 Technology choices

Technology	Design decision
Database	Should you use a relational SQL database or an unstructured NoSQL database?
Platform	Should you deliver your product on a mobile app and/or a web platform?
Server	Should you use dedicated in-house servers or design your system to run on a public cloud? If a public cloud, should you use Amazon, Google, Microsoft, or some other option?
Open source	Are there suitable open-source components that you could incorporate into your products?
Development tools	Do your development tools embed architectural assumptions about the software being developed that limit your architectural choices?

4.5 Technology issues

An important part of the process of designing software architecture is to make decisions about the technologies you will use in your product. The technologies that you choose affect and constrain the overall architecture of your system. It is difficult and expensive to change these during development, so it is important that you carefully consider your technology choices.

An advantage of product development compared to enterprise system development is that you are less likely to be affected by legacy technology issues. Legacy technologies are technologies that have been used in old systems and are still operational. For example, some old enterprise systems still rely on 1970s database technology. Modern systems may have to interact with these and this limits their design.

Unless you have to interoperate with other software products sold by your company, your choice of technology for product development is fairly flexible. Table 4.8 shows some of the important technology choices you may have to make at an early stage of product development.

4.5.1 Database

Most software products rely on a database system of some kind. Two kinds of database are now commonly used: relational databases, in which the data are organized into structured tables, and NoSQL databases, in which the data have

a more flexible, user-defined organization. The database has a huge influence on how your system is implemented, so which type of database to use is an important technology choice.

Relational databases, such as MySQL, are particularly suitable for situations where you need transaction management and the data structures are predictable and fairly simple. Relational databases support ACID transactions. Transactions guarantee that the database will always remain consistent even if a transaction fails, that updates will be serialized, and that recovery to a consistent state is always possible. This is really important for financial information where inconsistencies are unacceptable. So, if your product deals with financial information, or any information where consistency is critical, you should choose a relational database.

However, there are lots of situations where data are not well structured and where most database operations are concerned with reading and analyzing data rather than writing to the database. NoSQL databases, such as MongoDB, are more flexible and potentially more efficient than relational databases for this type of application. NoSQL databases allow data to be organized hierarchically rather than as flat tables, and this allows for more efficient concurrent processing of "big data."

Some applications need a mixture of both transactions and big data processing, and more of this kind of application will likely be developed in the future. Database vendors are now starting to integrate these approaches. It is likely that, during the lifetime of this book, efficient integrated database systems will become available.

4.5.2 Delivery platform

Globally, more people access the web using smartphones and tablets rather than browsers on a laptop or desktop. Most business systems are still browser-based, but as the workforce becomes more mobile, there is increasing demand for mobile access to business systems.

In addition to the obvious difference in screen size and keyboard availability, there are other important differences between developing software for a mobile device and developing software that runs on a client computer. On a phone or tablet, several factors have to be considered:

1. *Intermittent connectivity* You must be able to provide a limited service without network connectivity.
2. *Processor power* Mobile devices have less powerful processors, so you need to minimize computationally intensive operations.

3. *Power management* Mobile battery life is limited, so you should try to minimize the power used by your application.
4. *On-screen keyboard* On-screen keyboards are slow and error prone. You should minimize input using the screen keyboard to reduce user frustration.

To deal with these differences, you usually need separate browser-based and mobile versions of your product front-end. You may need a completely different decomposition architecture in these different versions to ensure that performance and other characteristics are maintained.

As a product developer, you have to decide early in the process whether you will focus on a mobile or a desktop version of your software. For consumer products, you may decide to focus on mobile delivery, but for business systems, you have to make a choice about which should be your priority. Trying to develop mobile and browser-based versions of a product at the same time is an expensive process.

4.5.3 Server

Cloud computing is now ubiquitous so a key decision you have to make is whether to design your system to run on individual servers or on the cloud. Of course, it is possible to rent a server from Amazon or some other provider, but this does not really take full advantage of the cloud. To develop for the cloud, you need to design your architecture as a service-oriented system and use the platform APIs provided by the cloud vendor to implement your software. These allow for automatic scalability and system resilience.

For consumer products that are not simply mobile apps, I think it almost always makes sense to develop for the cloud. The decision is more difficult for business products. Some businesses are concerned about cloud security and prefer to run their systems on in-house servers. They may have a predictable pattern of system usage, so there is less need to design the software to cope with large changes in demand.

If you decide to develop for the cloud, the next decision is to choose a cloud provider. The major providers are Amazon, Google, and Microsoft, but unfortunately their APIs are not compatible. This means you can't easily move a product from one to the other. The majority of consumer products probably run on Amazon's or Google's cloud, but businesses often prefer Microsoft's Azure system because of its compatibility with their existing .NET software. Alternatively, there are other cloud providers, such as IBM, that specialize in business services.

4.5.4 Open source

Open-source software is software that is freely available and you can change and modify it as you wish. The obvious advantage is that you can reuse rather than implement new software, thereby reducing development costs and time to market. The disadvantages of open-source software are that you are constrained by that software and have no control over its evolution. It may be impossible to change that software to give your product a "competitive edge" over competitors that use the same software. There are also license issues that must be considered. They may limit your freedom to incorporate the open-source software into your product.

The decision about open-source software also depends on the availability, maturity, and continuing support of open-source components. Using an open-source database system such as MySQL or MongoDB is cheaper than using a proprietary database such as Oracle's database system. These are mature systems with a large number of contributing developers. You would normally only choose a proprietary database if your product is aimed at businesses that already use that kind of database. At higher levels in the architecture, depending on the type of product you are developing, fewer open-source components may be available, they may be buggy, and their continuing development may depend on a relatively small support community.

Your choice of open-source software should depend on the type of product you are developing, your target market, and the expertise of your development team. There's often a mismatch between the "ideal" open-source software and the expertise that you have available. The ideal software may be better in the long term but could delay your product launch as your team becomes familiar with it. You have to decide whether the long-term benefits justify that delay. There is no point in building a better system if your company runs out of money before that system is delivered.

4.5.5 Development technology

Development technologies, such as a mobile development toolkit or a web application framework, influence the architecture of your software. These technologies have built-in assumptions about system architectures, and you have to conform to these assumptions to use the development system. For example, many web development frameworks are designed to create applications that use the model-view-controller architectural pattern.

The development technology that you use may also have an indirect influence on the system architecture. Developers usually favor architectural choices that use familiar technologies that they understand. For example, if your team has a lot of experience with relational databases, they may argue for this instead of a NoSQL database. This can make sense, as it means the team does not have to spend time learning about a new system. It can have long-term negative consequences, however, if the familiar technology is not the right one for your software.

KEY POINTS

- Software architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.
- The architecture of a software system has a significant influence on non-functional system properties, such as reliability, efficiency, and security.
- Architectural design involves understanding the issues that are critical for your product and creating system descriptions that show components and their relationships.
- The principal role of architectural descriptions is to provide a basis for the development team to discuss the system organization. Informal architectural diagrams are effective in architectural description because they are fast and easy to draw and share.
- System decomposition involves analyzing architectural components and representing them as a set of finer-grain components.
- To minimize complexity, you should separate concerns, avoid functional duplication, and focus on component interfaces.
- Web-based systems often have a common layered structure, including user interface layers, application-specific layers, and a database layer.
- The distribution architecture in a system defines the organization of the servers in that system and the allocation of components to these servers.
- Multi-tier client–server and service-oriented architectures are the most commonly used architectures for web-based systems.
- Making decisions about technologies such as database and cloud technologies is an important part of the architectural design process.

RECOMMENDED READING

"Software Architecture and Design" This excellent series of articles provides sound, practical advice on general principles of software architecture and design. It includes a discussion of layered architectures in Chapter 3, under architectural patterns and styles. (Microsoft, 2010)

https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658093(v%3dpandp.10)

"Five Things Every Developer Should Know about Software Architecture" This is a good explanation of why designing a software architecture is consistent with agile software development. (S. Brown, 2018)

https://www.infoq.com/articles/architecture-five-things

"Software Architecture Patterns" This is a good general introduction to layered architectures, although I don't agree that layered architectures are as difficult to change as the author suggests. (M. Richards, 2015, login required)

https://www.oreilly.com/ideas/software-architecture-patterns/page/2/layered-architecture

"What is the 3-Tier Architecture?" This is a very comprehensive discussion of the benefits of using a three-tier architecture. The author argues that is isn't necessary to use more than three tiers in any system. (T. Marston, 2012)

http://www.tonymarston.net/php-mysql/3-tier-architecture.html

"Five Reasons Developers Don't Use UML and Six Reasons to Use It" This article sets out arguments for using the UML when designing software architectures. (B. Pollack, 2010)

https://saturnnetwork.wordpress.com/2010/10/22/
five-reasons-developers-dont-use-uml-and-six-reasons-to-use-it/

"Mobile vs. Desktop: 10 key differences" This blog post summarizes the issues to be considered when designing mobile and desktop products. (S. Hart, 2014)

https://www.paradoxlabs.com/blog/mobile-vs-desktop-10-key-differences/

"To SQL or NoSQL? That's the Database Question" This is a good, short introduction to the pros and cons of relational and NoSQL databases. (L. Vaas, 2016)

https://arstechnica.com/information-technology/2016/03/
to-sql-or-nosql-thats-the-database-question/

I recommend articles on cloud-computing and service-oriented architecture in Chapters 5 and 6.

PRESENTATIONS, VIDEOS, AND LINKS

https://iansommerville.com/engineering-software-products/software-architecture

EXERCISES

4.1 Extend the IEEE definition of software architecture to include a definition of the activities involved in architectural design.

4.2 An architecture designed to support security may be based on either a centralized model, where all sensitive information is stored in one secure place, or a distributed model, where information is spread around and stored in many different places. Suggest one advantage and one disadvantage of each approach.

4.3 Why is it important to try to minimize complexity in a software system?

4.4 You are developing a product to sell to finance companies. Giving reasons for your answer, consider the issues that affect architectural decision making (Figure 4.4) and suggest which two factors are likely to be most important.

4.5 Briefly explain how structuring a software architecture as a stack of functional layers helps to minimize the overall complexity in a software product.

4.6 Imagine your manager has asked you whether or not your company should move away from informal architectural descriptions to more formal descriptions based on the UML. Write a short report giving advice to your manager. If you don't know what the UML is, then you should do a bit of reading to understand it. The article by Pollack in Recommended Reading can be a starting point for you.

4.7 Using a diagram, show how the generic architecture for a web-based application can be implemented using a multi-tier client–server architecture.

4.8 Under what circumstances would you push as much local processing as possible onto the client in a client–server architecture?

4.9 Explain why it would not be appropriate to use a multi-tier client–server architecture for the iLearn system.

4.10 Do some background reading and describe three fundamental differences between relational and NoSQL databases. Suggest three types of software product that might benefit from using NoSQL databases, explaining why the NoSQL approach is appropriate.