GLOBAL EDITION

Engineering Software Products

An Introduction to Modern Software Engineering

Ian Sommerville

ENGINEERING SOFTWARE PRODUCTS

An Introduction to Modern Software Engineering

Global Edition

Ian Sommerville

Pearson

Harlow, England • London • New York • Boston • San Francisco • Toronto • Sydney • Dubai • Singapore • Hong Kong
Tokyo • Seoul • Taipei • New Delhi • Cape Town • São Paulo • Mexico City • Madrid • Amsterdam • Munich • Paris • Milan

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.

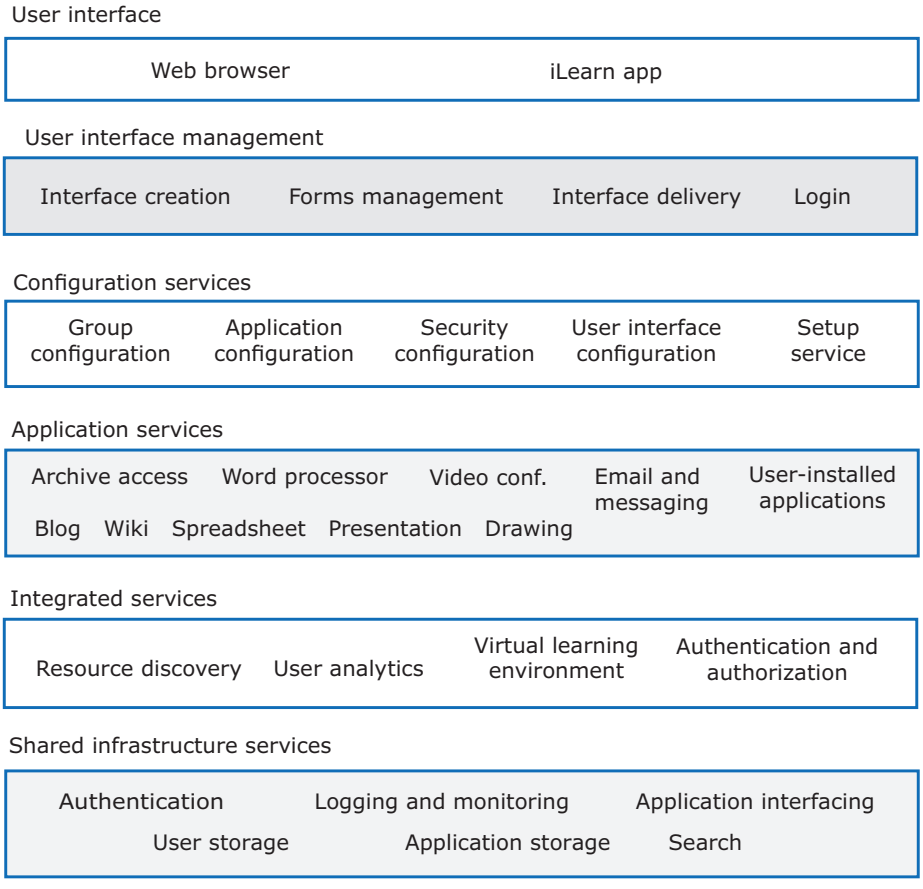
We assumed that only a minority of users would be interested in programming their own system versions. Therefore, we decided to provide a standard set of application services that had some degree of integration with other services. We anticipated that most users would rely on these and would not wish 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.

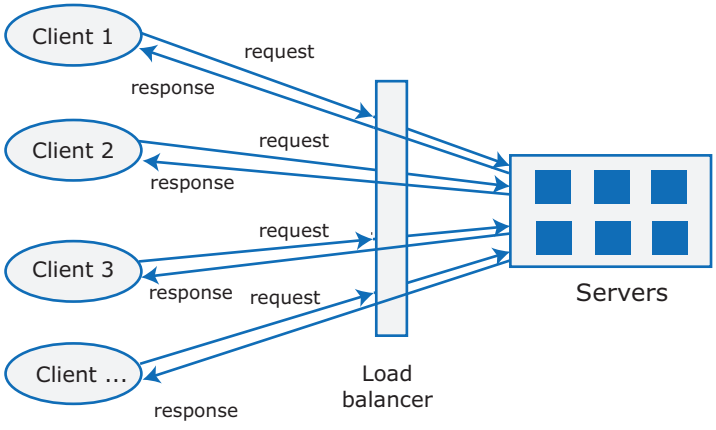
Another important technology-related decision is the interface technologies that you will use. This choice depends on whether you will be supporting browser interfaces only (often the case with business systems) or you also want to provide interfaces on mobile devices. If you are supporting mobile devices,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

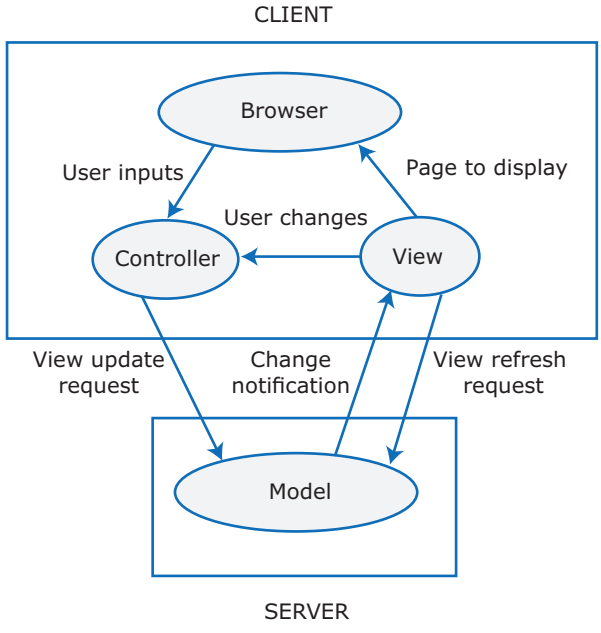
Client–server interaction is usually organized using what is called the Model-View-Controller (MVC) pattern. This architectural pattern is used so that client interfaces can be updated when data on the server change (Figure 4.13).

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.