Software Architecture and Standardization

4

An elaborate software infrastructure joins the equipment emphasized in Chapter 3 in providing many capabilities benefiting all networked applications.

ANALOGY: *In creating a new business, much existing infrastructure—such as real estate and telephone and transportation—is available and analogous to the computer and network equipment. A services infrastructure analogous to the software infrastructure includes package delivery, legal and accounting firms, and real estate property management. Incorporating these existing capabilities rather than building them from scratch makes it much easier to build the business.*

Like the equipment emphasized in Chapter 3, the software infrastructure has an architecture, but one that is quite distinct from the equipment. This software infrastructure is based on layering, in which new capabilities are incrementally added to existing capabilities. Like the equipment infrastructure, the software infrastructure is typically integrated from components supplied by different vendors. This makes it necessary for vendors to get their software to work together properly, and this is the role of a business process called standardization.

4.1 What Makes a Good Architecture

The concept of an architecture was defined in Section 3.1.1 on page 78 and applied to the decomposition of a networked computing system into hosts and network in Chapter 3. Recall that an architecture decomposes a system into subsystems (where those subsystems may be components if they are purchased without modification from an outside vendor), with specific functionality and interaction among subsystems. A good architectural design requires an appreciation for what distinguishes "good" from "bad" architectures and the criteria used in determining what is "good." Architecture follows some well-established principles that aid in understanding the software infrastructure. Application software is deferred to Chapter 6.

A major challenge in software design is complexity. Because software is not constrained by physical limits, its complexity tends to expand to the limits of the designer's ability to cope (discussed further in Chapter 6). This problem is accentuated for infrastructure software, because its design and development is spread over multiple vendors. Artificial design constraints have to be established to contain complexity, and that is one role of architecture.

Architecture is one phase in the design of the software—both the infrastructure (considered here) and application (considered in Section 3.4.2 on page 106)—after analysis and before implementation. Important inputs are functionality and performance requirements, and an outcome is the specification of the architectural subsystems sufficiently detailed to set about their implementation.

ANALOGY: *The framers of the U.S. Constitution (who were the architects of the U.S. government) first decided on the roles and limitations of the federal government. Taking this "functionality" into account, they decomposed the government into executive, legislative, and judicial branches. Next, they determined the responsibilities and powers of each branch and their interactions.*

4.1.1 Decomposition and Modularity

Decomposition—the first and most important phase of architecture—is a divide-and-conquer strategy that allows subsystems to be implemented individually and, ideally, even autonomously. It allows individual firms to participate in the design, manufacture, and deployment of the infrastructure without excessive coordination,

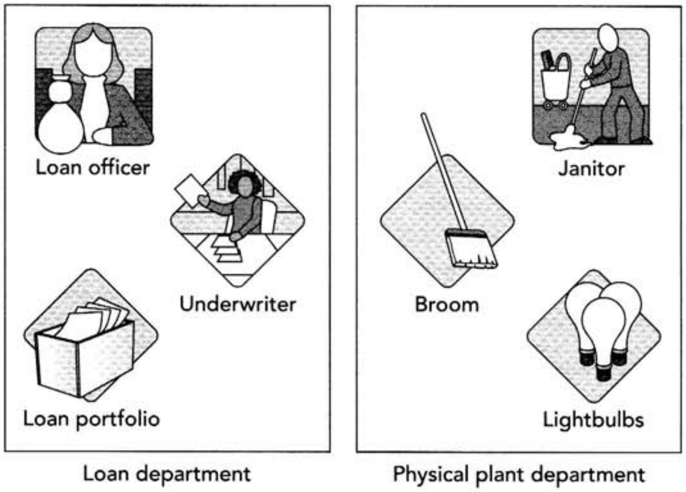

Figure 4.1 Partial decomposition of a bank.

and thus it affects the supplier industry organization as well as the technology.

The decomposition should be *modular,* which means the subsystems—called *modules*—have some special properties summarized in Table 4.1. The major property is *separation of concerns;* that is, the internal concerns of one module are mostly not of concern to other modules [Boo94]. This allows the modules to be designed and maintained relatively independently by different design groups or firms, with minimum interaction among them.

ANALOGY: *A bank is a system providing financial services to its customers. A partial decomposition of a bank is shown in Figure 4.1. It has departments with relatively little concern about the internal operation of others. The loan department focuses on managing its loan portfolio, underwriting (assessing the risk of a loan in the context of a portfolio), and issuing new loans, while the physical plant department maintains the buildings. There is interaction between departments, but it is less frequent and involved than activities within each department. For example, the loan department may report a dark room to the physical plant department but cares only*

Table 4.1 Some desirable characteristics of a modular architecture.

Property	*Desirable characteristics*	*Analogy*
Functionality	The modules are chosen as distinct functional groupings.	In government, establishing and enforcing laws and sitting in judgement of lawbreakers are distinct and well-defined functions.
Hierarchy	Each module can itself be a system, internally decomposed into modules (see Section 3.1.2 on page 79).	The executive branch of government is decomposed into agencies and departments, each of which has an internal modular structure.
Separation of concerns	The functional groupings incorporated within each module are strongly associated and weakly associated with functionality internal to other modules.	Law enforcement requires internal coordination, but its operations are of little concern to the legislature or judiciary.
Interoperability	Modules can successfully interact to realize the higher purposes of the system.	Law enforcement has well-defined procedures for bringing alleged lawbreakers before the judiciary, which results in their successful prosecution.
Reusability	Modules are defined, implemented, and documented independently of a specific system, so they can be reused in other systems.	Some modules of the U.S. government structure have been adopted by other countries, without the need to adopt them all.

that light is restored and not how this happens. The physical plant department concerns itself with details, such as diagnosing the problem.

EXAMPLE: *The architecture of a computer is an example of modularity. Figure 4.2 presents a simplified view, including major modules:*

- *A* processor *executes a program.*
- *The* memory *stores program instructions and data currently being used.*
- *The* storage, *such as magnetic disks, CD-ROMs, etc., keeps massive amounts of data, programs, etc.*
- *The* network adapter *connects the host to a network.*

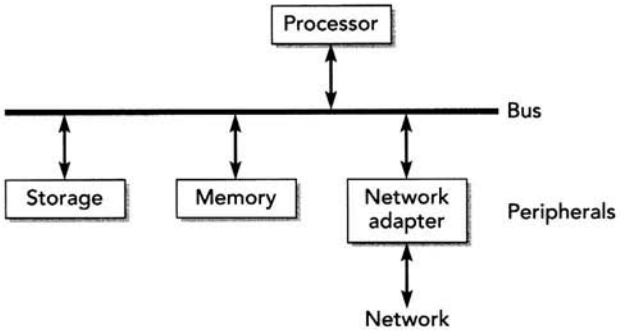

Figure 4.2 The simplified modularity in a computer system.

The storage and network adapter are called peripherals, *because they assist the processor, and are connected to the processor by a* bus *(a very high speed connection shared by all the modules). Each of these modules has lots of internal activity, but their interaction with others is relatively straightforward. Modules can be designed and manufactured by different firms because of that well-defined and well-documented interaction; thus, these modules are components (see "Subsystems and Components" on page 80).*

4.1.2 Granularity and Hierarchy

Granularity determines the number of modules and the range of functionality of each. The architecture designer can choose a *fine* granularity, with many small modules, or a *coarse* granularity, with a few large modules. Hierarchy—meaning modules are themselves composed of internal modules—avoids defining a *single* granularity (see Section 3.1.2 on page 79). This "decomposition within modules" architecture allows the system to be viewed at different granularity, as appropriate.

ANALOGY: *In the bank organization of Figure 4.1, each of the two departmental modules has an internal decomposition into smaller modules. For example, the loan department is internally composed of a receptionist, an underwriter, a loan officer, etc. As there may be multiple underwriters and loan officers, there are finer-granularity modules associated with the loan officers (the "loan service* *group") and all underwriters (the "risk management group"). At a coarser granularity, the bank is decomposed into "customer services" and "business management" divisions, in which the former groups modules dealing with customers (loans, withdrawals, deposits, etc.) and the latter encompasses internal functions (accounting, property management, physical plant, etc.).*

EXAMPLE: *The single computer, whose internal composition into modules is shown in Figure 4.2, is itself a module (called a host) in the larger client/server architecture discussed in Chapter 3. The overall hardware architecture is thus a two-level hierarchy.*

4.1.3 Interfaces: The Module's Face to the World

Each module must interact with others to accomplish the higher purposes of the system. The architecture designer should be quite thoughtful about this interaction to ensure that the system operates correctly under all circumstances and is flexible enough to accommodate future change. This is assisted by defining an *interface* to each module. The interface is the view that one module presents to others, encompassing everything that other modules must know to interact with it. It has a second purpose, which is to guide the module implementers.

EXAMPLE: *In the bank example of Figure 4.3, the interface to the loan department expects certain standard actions, such as a request for a loan (from a customer) and the submission of a completed loan application (from the customer). Each request has predetermined responses, such as returning a blank loan application and responding yes or no to the loan request.*

More generally, each module typically has a standard "menu" of actions it will take, where each action has a set of *parameters* and *returns*. The interface description includes all actions a module is prepared to take, together with a definition of the parameters and returns of those actions.

A general approach to module interaction is shown at the bottom of Figure 4.3. That interaction includes a series of *invocations* of

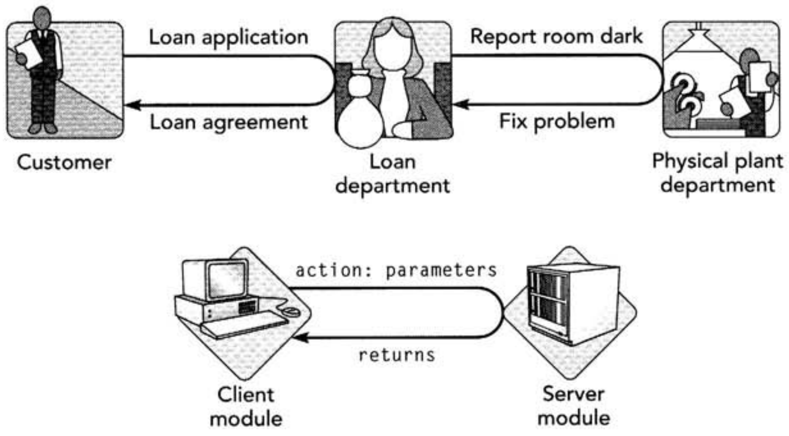

Figure 4.3 Examples of module interfaces in the bank example.

actions defined at the module interfaces (only one such invocation is shown). In each invocation, the module invoking the action is called the client module, and the module whose action is invoked is called the server module. (Note that the terminology was used to describe the role of hosts in Chapter 3.) Rarely does a module act exclusively as a client or server; at different times, it is each.

EXAMPLE: *In Figure 4.3, when a customer makes a loan application ("invokes the loan application action"), she is a client of the bank, and the bank loan department is a server to her.*

4.1.4 Abstraction

People use *abstraction* to make complicated things easier to deal with. Its proper use in architecture design makes that architecture more transparent and flexible to future change.

Abstraction is concisely defined as "generalization; ignoring or hiding details." In the context of architecture, abstraction is used to simplify the perspective of a module as viewed through its interface, focusing on the important overall goals of the system and avoiding becoming mired in a clutter of unnecessary details (see the sidebar "Example of Abstraction: The Flora" for another example).

Example of Abstraction: The Flora

Physical and social scientists abstract complicated and interdependent natural and social systems. To make the study of complicated systems feasible, they focus on the aspects most relevant to the investigation at hand, ignoring other less germane details. This is not limited to scientists; for example, consider the following perspectives on the flora taken by different occupations:

- The botanist classifies plants based on evolutionary family dependencies.
- The master chef studies a plant's taste and smell, whether it is edible or poisonous, how long it takes to cook, etc.
- The gardener is concerned with the adult size of the plant, what type of soil and climate conditions it favors, how much fertilizer it needs, etc.
- The pharmacologist looks for medicinal effects in each plant.

Although there is overlap and dependency, each profession finds germane a different aspect of the flora. Each is abstracting the flora for its own purposes.

EXAMPLE: *To a top executive of the bank, the loan department is a module that makes money for the bank by accepting loan applications and issuing loans likely to be repaid. This abstract perspective—when turned into reality by the manager setting up the loan department—has to be made concrete by setting up detailed steps for the loan department to determine whether a loan is a good bet. Those details may actually change over time—based on experience—without affecting the abstract view taken by the executive.*

As this example suggests, abstraction and "management hierarchies" in organizations go hand in hand. Each higher layer in the management hierarchy takes an increasingly abstract view of the organization's architecture. When it comes to actually setting up the lower-level departments, there are a plethora of details handled internally.

EXAMPLE: *The manager of the bank's loan department is responsible for determining the detailed processes within that module for achieving the abstracted vision of that department held by top management. She may set up a multistep loan approval process, such as running a credit history on the loan applicant, appraising any collateral offered by the loan applicant, seeking the advice of an experienced underwriter, etc. Meantime, the higher management views the department in abstract terms like "quarterly profit and loss."*

Abstraction is effective because it enables issues of importance to a system as a whole to be considered without being obscured by distracting details. At lower levels of hierarchy, those details are dealt with, but in a smaller context constrained by the higher-level abstractions. An important issue is choosing the appropriate abstractions, using them to make the system simpler and easier to deal with, but not so simple as to be unrealistic. As Albert Einstein stated, "Everything should be made as simple as possible, but no simpler."

4.1.5 Encapsulation

An architecture focuses on the external behavior of the modules—as manifested by their interfaces—and how they interact. Implementers must take this interface and determine the internal design of the modules. An important architectural and implementation tool is *encapsulation*, which ensures internal details are invisible and inaccessible at the interface. This avoids other modules becoming dependent on internal details, making the system more difficult to change.

EXAMPLE: *In an abstract interface to the loan department, the customer is not responsible for invoking the steps of the loan approval process—that is entirely the business of the loan department. Encapsulation goes further and ensures these steps are invisible to the customer. For example, encapsulation ensures there are no other actions at the interface that provide visibility into the outcome of individual approval steps (such as the credit history report or the advice of the underwriter).*

Abstraction and encapsulation are complementary. Both seek separation of concerns (see Table 4.1 on page 116), the former by simplifying the external view and the latter by dogmatically enforcing abstractions by hiding internal details from the interface. Encapsulated details can be safely changed without affecting other modules.

EXAMPLE: *Which credit bureau the loan department consults to obtain a credit report on the loan applicant is encapsulated so that the bureau can be changed at any time without affecting the customer. Indeed, the entire credit bureau step is encapsulated so that it can be eliminated entirely if it proves ineffective.*

4.1.6 Modularity and Interfaces in Computing

For the remainder of the book, the principles of architecture design will be applied to computing (although they apply in other contexts as well). First, it is helpful to reflect briefly on the meaning and implications of architecture in computing.

Hardware/Software Dichotomy

The computer embodies many ideas, but the most powerful is programmability: Unlike earlier products, the functionality of a computer isn't determined at the time of manufacture, but is added later by software.

This makes the computer almost infinitely extensible. What can be accomplished with a computer is limited primarily by the programmer's imagination (and pragmatic constraints such as complexity and cost) rather than physical limitations.

Software Architecture

A software program running on a computer tells it what to do each step of the way. Although it isn't subject to the same physical limitations as the hardware, it is important to *impose* an architecture on software. The two most important reasons are to manage the inherent complexity (see Chapter 6) and to coordinate software provided by a number of competing and complementary firms (which is considered in this chapter). Software should be designed to be modular—decomposing it according to functionality—so that different firms or programming groups can implement different modules with minimum dependence among them. The principle of modularity as a "separation of concerns" is critically important in software.

Hardware and Software Interfaces

Interfaces are an integral part of the computing world for both hardware and software (see Section 4.1.3 on page 118). A hardware interface is a physical wire or fiber and connector and the precise definition of the electrical or optical signals it carries.

EXAMPLE: *In computers, the assorted jacks on the back (serial port, parallel port, monitor jack, power plug, etc.) are examples of interfaces. Each is associated with a set of physical (number of pins, geometry, etc.), electrical (voltage, etc.), and logical (order and meaning of bits) specifications.*

In software, an interface is the boundary between two software programs, or the boundary between two modules within the same software program. Its purpose is to allow the modules or programs to interact to accomplish some higher purpose, while encapsulating the inner workings, denying other modules access to them. The form of this interface is similar to interfaces found within organizations (see Section 4.1.3 on page 118). In particular, a software module interface consists of a set of actions, each action having a set of parameters passed to the module that customizes that action, and a set of returns from the module reflecting the results of that action. The action may change data within the modules as well as return values.

EXAMPLE: *It might be useful within a larger software system to have a module that provides the same capabilities as a pocket calculator, such as adding, subtracting, multiplying, dividing, taking square roots, etc. Its interface could be similar to a pocket calculator, except there aren't physical keys and a physical display. The equivalent functions are invoked by other modules using actions like "add," "subtract," etc., with appropriate parameters consisting of the numbers to be added or subtracted.*

This calculator module can serve other programs needing numerical calculations. Significantly, the calculator interface reveals nothing about how the calculator performs these functions—these details are encapsulated, as they are in a real calculator.

4.2 Architecture of the Software Infrastructure

The architectural concepts, including decomposition, modularity, hierarchy, granularity, interface, abstraction, and encapsulation, aid the understanding and appreciation of the software infrastructure supporting networked applications. They are also applied to application design in Chapter 6. The more detailed functionality of the infrastructure is deferred to Chapter 7 and later chapters.

4.2.1 Goals of the Infrastructure

Well-principled architecture design is one goal of the infrastructure—in part to contain complexity—but there are others, such as

- Minimizing the cost and maximizing the performance (see Chapter 10).

- Minimizing the effort required to develop and maintain new applications, in part by including capabilities in the infrastructure required by a wide range of applications.
- Providing capabilities to support the operation of the system and contribute to its trustworthiness and reliability (see Chapter 8).

The layered architecture described next was not designed top-down by a single individual or organization. It is the result of the evolution of computing systems over many years, an evolution almost Darwinian in nature because it has involved many hundreds of hardware and software suppliers and tens or hundreds of thousands of individual contributors. Standardization plays an important role in coordinating these many players (see Section 4.3 on page 132). Many nontechnical considerations that help explain the technological trajectory are discussed in Chapter 5.

4.2.2 Layering

The form of modularity seen in the software infrastructure, at the top level of hierarchy, is *layering*, similar to the rings of an onion. The modules composing the infrastructure are layered, one "above" the other, where the terms "above" and "below" are interpreted logically, not physically. Each layer utilizes the capabilities of the layers below it and adds capabilities of its own to provide to the layers above it. Thus, layering is a way to achieve additional capability to adding infrastructure, making use of what already exists rather than building from scratch.

The software and hardware follow this modularity; in particular, the software is thought of as "riding on top of" the hardware, utilizing hardware capabilities to run programs, but abstracting the hardware's detailed characteristics from the application and user (this is the role of the *operating system* layer).

The *layering principle* sets out the constraints (see Figure 4.4):

- Each layer acts as a server to the layer above, providing actions whose implementations are encapsulated.
- Each layer is a client to the layer below, utilizing its available actions in the course of providing services to the layer above it.

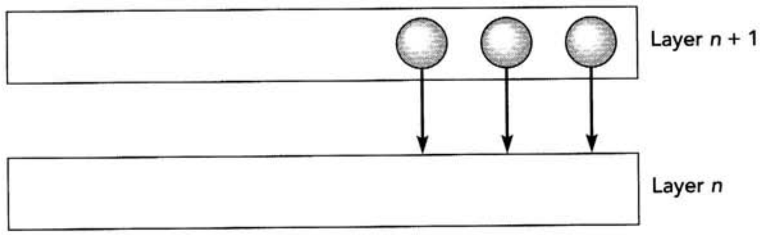

Figure 4.4 The layering principle in software/hardware modularity.

- Each layer is permitted to interact with only layers immediately above and below. Thus, each layer serves to hide (encapsulate) the layers below it from the layers above it.

Functionally, the idea is to provide increasingly abstract and specialized services at each higher layer. Each layer is thus "simplifying, by hiding unnecessary detail" the layer below.

ANALOGY: *Consider a company that manufactures cyberwidgets. The architecture of the company defines four layers—supply, receiving, inventory, and assembly—and a building designed around these modules. The inventory layer is a module that is a server to the assembly layer and a client of the receiving layer, etc. The building has three floors, one for each of the receiving, inventory, and assembly modules. The supply layer encapsulates all the suppliers of parts assembled into cyberwidgets and is external to the building, but interacts through the trucks and drivers they send. The interaction among modules follows the layering principle in that the module corresponding to each floor (one layer) makes direct use of the capability of the floor below, but no others. The four layers are described in Table 4.2.*

The architecture of the building can support these layers by including means for each floor to make requests of the floor below (say, by an intercom system) and means to convey parts to the floor immediately above (such as conveyer belts) in response to requests.

This analogy has other features similar to the computer infrastructure. For example, specific functions analogous to "supply," "receiving," and "inventory" are required in networked computing

Table 4.2 A layered architecture for the manufacturer of cyberwidgets.

Layer	*Functionality*	*Interaction with layer below*
Assembly (third floor)	Assembles parts into finished cyberwidgets.	Requests specific parts as needed for assembly.
Inventory (second floor)	Stores parts awaiting assembly.	Indicates which parts are needed (in danger of starving assembly layer) or in excess supply (filling up allotted space). Requests and stores any parts that have arrived in receiving.
(Receiving (first floor)	Coordinates the supply of parts, receives parts, and pays suppliers.	Orders and receives parts, unloads, counts, and authorizes payment.
Supply (external)	Manufactures and conveys parts to be assembled into cyberwidgets.	There is no layer below.

(see Chapter 7). The coordination of supplier with assembly illustrates a problem in computing called *flow control*, described in Chapter 11.

The Layers in a Computing Infrastructure

Figure 4.5 illustrates the major infrastructure layers in a networked computing system. The functionality of each layer is summarized in Table 4.3.

How do the layers interact to get things done? Figure 4.6 shows two hosts participating in a networked application, each host executing a piece of that application and these pieces collaborating by communicating data through the network (as indicated by the arrow). The application does not access the network directly—this would violate the layering principle—but rather it involves the middleware and operating system layers. The middleware presents a standard interface to the application independent of the operating system and networking technology.

The middleware logically spans the two hosts (shown by dotted lines) because its goal is to hide the details of the operating system and network, including the distribution across hosts.

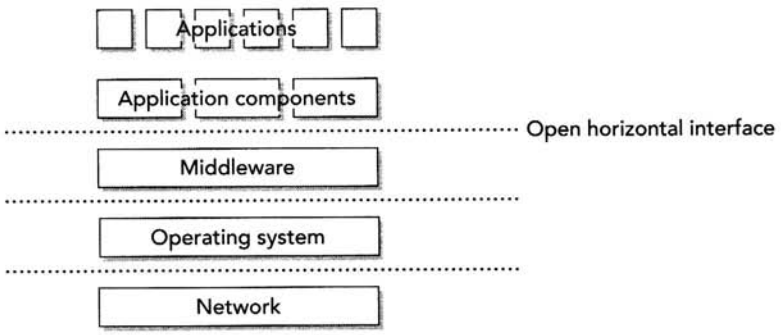

Figure 4.5 A simplified layered architecture for networked computing software infrastructure.

Table 4.3 The major layers in a computer infrastructure.

Layer	*Function*	*Analogy*
Application (Chapter 6)	Provides specialized functionality directly needed by a user or organization (e.g., electronic commerce, information retrieval, or collaboration).	A firm is in a particular line of business (e.g., automobile manufacture) and defines various processes tailored to the operation of that business (e.g., assembly lines).
Application components (Chapter 6)	Specialized modules incorporated by many applications and purchased as a product from an outside company.	All automobile manufacturers buy components, such as tires and batteries, from common suppliers.
Middleware (Chapter 9)	Hides the heterogeneity and distribution of operating system and network from the application. Also provides capabilities useful to a wide range of applications.	Professional services, such as accounting, law, private investigation, etc., benefit all firms.
Operating system (Chapter 10)	Manages and hides the details of resources such as storage and printing. Also manages the details of interhost communications.	Resource management services, such as real estate, janitorial, and gardening, are useful to firms.
Network (Chapter 11)	Provides communication of data from one host to another.	To support interaction among its different locations, a firm uses telephone and overnight package delivery companies.

A Layered View of the Life and Social Sciences

At the risk of serious oversimplification, both the life and social sciences can be viewed in layers. The biological sciences draw upon understanding of physical phenomena from physics and chemistry and can be viewed as layered upon them. Physiology builds on biology to understand the overall organism, and medicine in turn builds on physiology. Of course, each "layer" adds substantial understanding to that supplied by lower "layers."

Similarly, the social sciences begin with an understanding of individuals in psychology and linguistics, and sociology adds understanding of the behavior of groups of individuals. Economics, political science, and law—which deal with the organization of commerce and society for higher purposes—are the "architects" of the social sciences.

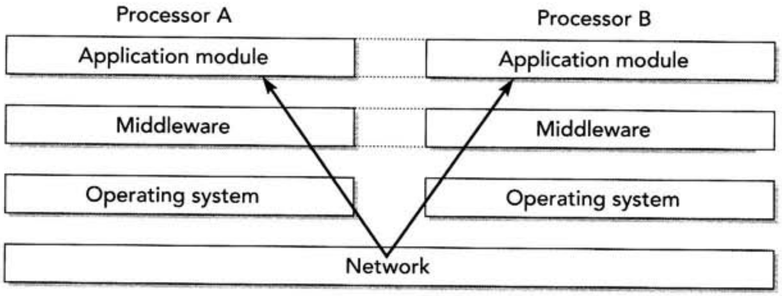

Figure 4.6 An example of application modules communicating through layers.

ANALOGY: *The mailroom in a large firm (analogous to middleware) provides workers with an abstracted letter and package delivery service. It relies on the post office and package delivery companies, or sometimes it may deliver a package directly inside the building. Those details are hidden (encapsulated) from mailroom clients.*

A top-down approach is followed in this book: Each layer is examined successively—starting with the application in Chapter 6—abstracting the layer below and appreciating what services it provides to the layer above. Much detail is ignored, focusing on issues important from an application perspective. This approach is quite similar to how computer systems are actually designed and implemented. Typically, in any given host, the layers are purchased from different companies, and it all works because of the layered modularity.

EXAMPLE: *Intel focuses on microprocessors, Compaq on desktop computers, Microsoft on operating systems, and Iona on middleware. Each is focused on one layer of the infrastructure, providing an interface promised to layers above.*

Data and Information in Layers

In Table 2.1 on page 18, data was described as "a collection of bits representing information." The infrastructure—both hardware and software—concentrates on storage and communication of data. On the other hand, what the application presents to the user is cer-

Table 4.4 How layers deal with data and information.

Structure and interpretation of data	*Storage*	*Communication*
The lowest layers deal with data in its most primitive form: a collection of bits. This portion of the infrastructure does not interpret those bits in any way.	The file system stores and retrieves bits without interpretation (see "File System" on page 91).	The network communicates packets and messages, both of which are presumed to contain bits without interpretation (see "The Network" on page 86).
	The operating system layer, which manages both storage and communication of data, usually assigns no interpretation to the data it manipulates on behalf of the application.	
The middle layers presume some structure for the data. Typically data is structured into numbers, character strings, and compositions of these basic types. In some cases, additional structure may be presumed.	A relational DBMS presumes that data is structured into tables (rows and columns of basic types) (see "Shared Data Tier: Database Management" on page 94).	The middleware layer communicates a structured composition of basic types (see Chapter 9).
The application assigns additional interpretation to the data.	Within the application, data is interpreted within the specific application context (for example, a number may be interpreted as a bank account balance).	

tainly information. The question then arises: Where and how is data turned into information? A rough answer is that this happens in the layers, where each layer adds structure and interpretation to the data it obtains from the layer below. As a rule, the infrastructure tries to make minimal assumptions about the structure of data, so that it can benefit a wide range of applications. However, moving into the middle layers, structure is added to the data so that more specific and useful facilities can be provided. The specific structure typically assigned to the data in different layers is described in Table 4.4.

The changing view of (often the very same) data as it moves through the infrastructure layers is at the heart of the separation of concerns that underlies the modularity of layering. Each layer avoids attaching any more structure and interpretation to the data it manipulates than is necessary to realize its functionality, thereby reducing its mutual dependence on other layers, including the application.

With this in mind, a definition can be assigned to the term "information" from the infrastructure and software perspective. If data is a collection of bits representing information, then information can be defined as structure and interpretation attached to data.

The Horizontal Layer Interface

Given the layered architecture, a key aspect of the infrastructure is the *horizontal layer interface*—illustrated by the dashed lines in Figure 4.5—which defines how each layer interacts with the layer below. Each layer interface is carefully documented, informing the layer above precisely how to invoke its services. One aspect of this interface is a presumed structure for data that passes between layers.

ANALOGY: *In the manufacturing example of Table 4.2 on page 126, the receiving floor provides a set of actions, such as "send next part that has arrived" and "obtain more parts with this number."*

These horizontal interfaces are said to be *open* when they are publicly available and not encumbered by intellectual property protections (see Chapter 5). Thus, any vendor is free to design, implement, and sell software that builds on an open interface without fear of violating legal protections, assuming the vendor possesses sufficient documentation to do so. Open interfaces not only enable different layers of the infrastructure to be designed (and manufactured, in the case of hardware) by complementary vendors but also support competition among vendors at each layer.

The Spanning Layer

The layered architecture just described is actually a simplification of reality. In practice—as a concession to competition in the industry—the infrastructure has some horizontal structure.

EXAMPLE: *In Figure 4.7, the layers are divided into modules at the granularity of an individual host. This heterogeneity arises not from the desire to have radically different functionality on different*

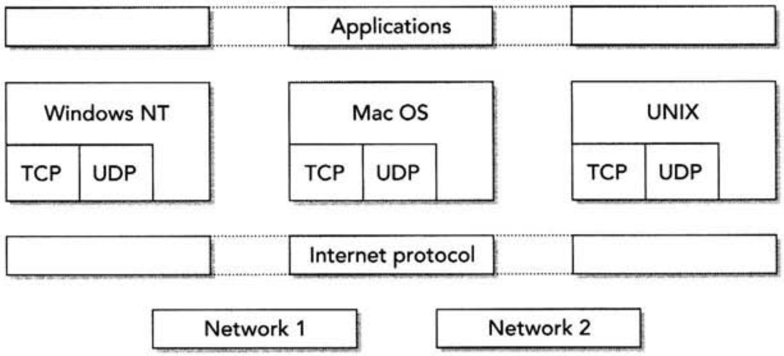

Figure 4.7 In reality, the architecture has structure in the horizontal as well as vertical direction.

hosts, but from the reality that different suppliers in the marketplace are providing products similar in functionality but distinct as to details. The two layers shown in Figure 4.7 with this characteristic are the operating system (where there are several major OSs in the marketplace, including Windows 95/NT, MacOS, and UNIX—see Chapter 10) and the network (where there are a number of different technologies, such as Ethernet, wireless, asynchronous transfer mode, etc.—see Chapter 11).

Layers that are homogeneous in the horizontal direction, and can be assumed to be virtually ubiquitous on all computing platforms, have special significance because they divide the infrastructure into quasi-independent components that can be developed and advanced separately and can hide any heterogeneity below. Further, such layers provide a large existing market to vendors selling products at the layers both above and below and thus attract investment and competition. A layer with these characteristics is called a *spanning* layer [Cla97]. The following examples are illustrated in Figure 4.8.

EXAMPLE: *The internet protocol (IP) layer provides communication services to applications using a variety of network technologies (see Chapter 7). It is the foundation of the internet protocols and has reached such wide acceptance among consumers, universities, and many companies that it is virtually a spanning layer.*

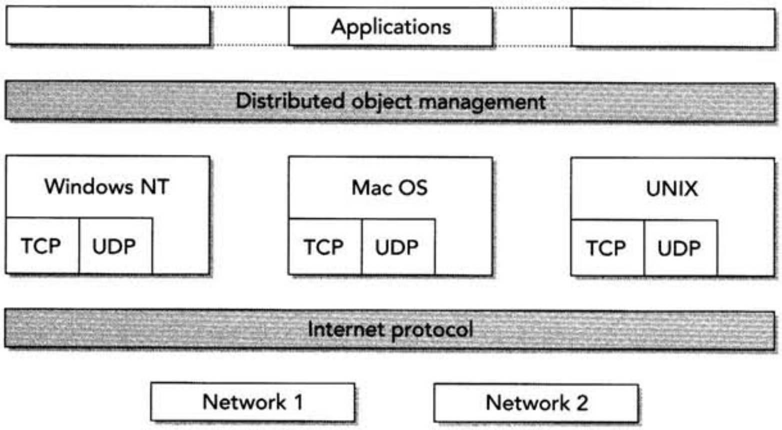

Figure 4.8 A spanning layer (shaded above) is uniform in the horizontal direction and almost ubiquitous.

> *The heterogeneity across different operating systems presents a problem to application developers. They must produce different versions for each operating system, creating a need for a spanning layer above the operating system. One candidate is* distributed object management *(DOM) (see Chapter 9).*

How Layers Enable Business Applications

Evolving from departmental to enterprise and cross-enterprise applications (see Section 2.6 on page 52) creates difficult technical challenges. The legacy client/server departmental applications created an obstacle to data integration across an enterprise because they proliferated heterogeneous infrastructure and applications. Similar problems arise in social applications (see Section 2.3 on page 19), particularly those serving interest groups and the citizenry, where the groups are so large that users are not coordinated. Fortunately, these challenges can be addressed by the layers described in Table 4.3. The challenges and technologies addressing them are summarized in Table 4.5.

4.3 Standardization

The modular layered architecture and open horizontal interfaces provide a way to coordinate vendors of infrastructure and application software and encourage competition in the industry. However,

Table 4.5 Challenges to integration of data across an enterprise.

Problem (layer addressing this problem)	*Nature of the problem*	*Solution*
Communication of data (network)	Numerous networking technologies, such as Ethernet, token ring, asynchronous transfer mode, etc., use different ways of transferring data.	The internet—since IP is a spanning layer—allows data to be communicated across heterogeneous networks by subsuming and interconnecting them (see Chapter 11).
Representation of structured data (middleware)	Different computer systems represent standard data values (e.g., character strings and numbers) in terms of bits in different ways (see the sidebar "Any Information Can Be Represented by Bits" on page 10).	Transparently to an application, the representation of data can be automatically converted when communicated from one host to another (see Chapter 9).
Interpretation of data (middleware)	The application must consistently interpret data values. For example, a character string might represent a person's name rather than his address, or a number might represent a person's zip code rather than her telephone number.	The structure and interpretation of data can be described in mutually agreed-upon ways using metadata (see "Assistance from the Author or Publisher" on page 43) or through interface specifications (see Chapter 9).

they raise a daunting problem: At each horizontal interface, the products of different suppliers must be *interoperable*—they must work together correctly. To achieve this, there must be a single standard interface definition, so that each product at one layer is interoperable with all products at the layers above and below. The solution is to standardize the interface, through the process of standardization. A *standard* is a specification generally agreed upon, precisely and completely defined, and well documented, so that any supplier can implement it.

ANALOGY: *Standardization is common in many industries and professions. For example, standards are set for the legal profession by the Uniform Commercial Code and for the accounting profession by the Financial Accounting Standards Board.*

Standardization within Applications

Standardization applies not only to infrastructure but also to some applications, particularly those that cross enterprise boundaries (see Section 2.6.3 on page 64).

EXAMPLE: *EDI (see "Electronic Data Interchange" on page 67) was stimulated by a standard, ANSI X12 [Kee97]. X12 specifies the formats for the exchanges of many standard business messages, including the invoice (X12-810), tax information reporting (X12-826), purchase order (X12-850), notice of employment status (X12-540), mortgage appraisal request (X12-261), and many others. Without ANSI X12 (and an equivalent European standard, EDIFACT), EDI would be far less prevalent.*

The significance of standards like ANSI X12 is that many businesses can exchange messages without prior coordination or negotiation of formats. They can purchase rather than develop software to generate and interpret messages particular to specific business partners. Another example of application standardization is digital payments (see Chapter 8).

A standardized interface offers more value to users and organizations because it empowers them to mix and match products from different vendors complying with the standard. There are thus market pressures on suppliers to create and adopt standardized interfaces. The dynamics of standardization are influential in determining the direction of computing as well as the winners and losers among vendors. Some economic foundations for standardization are discussed in Chapter 5.

4.3.1 Reference Models and Interfaces

Recall that an architecture defines decomposition, functionality, and interaction. In order to standardize interfaces, the decomposition and functionality have to be determined, so these are the first issues in standardization. Together they constitute a *reference model*, which is an overall definition of modularity—including the location of standardized interfaces—and the decomposition of functionality.

ANALOGY: *The human heart and circulatory system, as illustrated in Figure 4.9, define a decomposition into lungs, heart, body, and veins and arteries that carry the blood between them. The simple connectivity is defined in the diagram. Interfaces might specify the constituents of the blood, the rate of heartbeats, the blood pressure, etc.*

The layered architecture of Figure 4.5 is a useful reference model. After agreement on a reference model, the interfaces among the modules can be standardized, including (in the case of software) a set of actions, parameters, and return values, which are called *formats*. In addition, there are often constraints or expectations related to the sequence of invocation of actions, which are called *protocols* (see Chapter 7). Once a reference model is established, the standardization process focuses on the detailed specification of interfaces, including formats and protocols. A goal is to leave room for vendors to differentiate their products based on the standards, for example, with proprietary extensions.

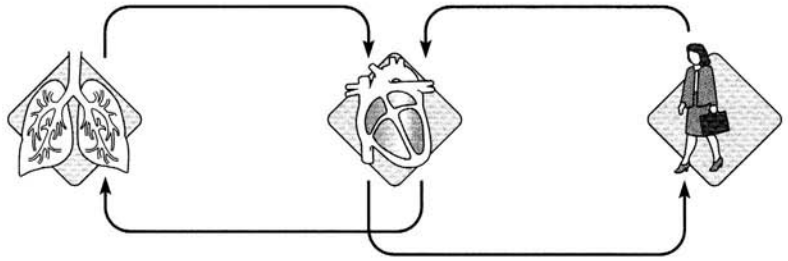

Figure 4.9 The circulatory system as a reference model.

4.3.2 Organization of the Standardization Process

Any formal standardization process requires a recognition of need by a standards body, industry organization, or government. A *standardization body* is an organization set up for the express purpose of promulgating standards (see the sidebar "International Organization for Standards (ISO)" for an example). The standards process also requires the commitment of monetary and human resources by a set of participating companies. A standards process may produce a single standard and dissolve, but more often there is an ongoing process of refinement and extension.

EXAMPLE: *The Internet requires complementary technologies and the coordination of a set of hardware suppliers (such as Cisco, 3Com, and Bay Networks), service providers (such as MCI and AT&T), and application suppliers (such as Netscape and Microsoft). These companies, together with university researchers, cooperate in a process of continuous refinement through the IETF (see the sidebar "Internet Engineering Task Force (IETF)").*

An increasingly popular approach in computing is a *technology web*, which is a set of companies coordinating complementary technologies without a formal process. (This isn't related to the Web information access application, except that the companies enhancing the Web are themselves a technology web.)

EXAMPLE: *The desktop personal computer is based on Intel microprocessors, the Microsoft operating system, various hardware suppliers, and various application software vendors—the Wintel*

International Organization for Standards (ISO)

ISO is an international, nongovernmental federation of national standards bodies from over 100 countries, one from each country (the American National Standards Institute (ANSI) is from the United States). Its stated role is to "promote the development of standardization and related activities in the world with a view to facilitating the international exchange of goods and services." The technical work is carried out in a hierarchy of some 2,700 technical committees, subcommittees, and working groups with representatives from industry, research institutes, government authorities, consumer bodies, and international organizations. They come together as equal partners in the resolution of global standardization issues.

ISO standards are not always successful in the marketplace. For example, the Open Systems Interconnection (OSI) was an elaborate standard for networking protocols that has been supplanted by the internet protocols (see the sidebar "Internet Engineering Task Force (IETF)").

Internet Engineering Task Force (IETF)

The IETF is a "large open international community of network designers, operators, suppliers, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet." The internet technologies are actually in direct competition with the OSI standards promulgated by ISO (see the sidebar "International Organization for Standards (ISO)"), but they have won hands-down in the marketplace. The reasons are several. The Internet standards have been developed incrementally from simple beginnings, always accompanied by a working implementation funded largely by government research budgets. OSI, in contrast, attempted a complete "paper" design without implementation. The Internet standards reached the marketplace much faster and were distributed free, as part of the Berkeley UNIX system. The result is that the Internet standards had already developed a large suite of applications before the OSI standards could be completed. This is an illustration of the importance of "time-to-market" and "winner-take-all" effects (see Chapter 5).

Table 4.6 Two types of standards.

Type	*Description*	*Examples*
De facto standard	A technology so commonplace that it is a standard in *reality*, even if not recognized by any formal body. It can be established in a couple ways: market power and voluntary cooperation. With *market power*, some product categories have winner-take-all effects resulting over time in a dominant solution (see Chapter 5). With *voluntary cooperation*, companies who recognize the need for interoperability voluntarily work together to recommend standards (see the sidebar "Object Management Group").	Market power: Windows operating system and the Hayes command set; voluntary cooperation: internet protocols (Chapter 7), Java (Chapter 9), and CORBA (Chapter 9).
De jure standard	A standard established by a formal process organized by government, an industry association, or standardization body. It may actually be mandated by law.	ISDN telephone interface; X.500 directory service; GSM digital cellular telephone.

technology web. Intel is particularly active in promulgating hardware interface standards through its Architecture Laboratory, and Microsoft is similarly active in software interface standards.

While the formal standards process is at times slow, technology webs support a rapid continual technology refinement. A technology web is typically limited to a small set of suppliers (often only one supplier for each of the complementary technologies), while most standards processes welcome all comers.

Two distinct types of standards—*de facto* and *de jure*—are described in Table 4.6. The computer industry moves so quickly that the de facto standard is increasingly popular.

4.3.3 Control and Enforcement of Standards

Issues with commercial, legal, and political implications surround the control and enforcement of standards. These issues heavily influence the competitive outcomes in the industry, and standards are increasingly a battleground for supremacy [Sha98].

As described in "The Horizontal Layer Interface" on page 130, many standards are *open* (publicly documented and unencumbered by intellectual property restrictions). For them, the primary "enforcement" is the marketplace, which favors products complying with the standard, in part because they are interoperable with complementary products.

A de facto standard may in principle be available for use by anyone, but there is a dominant proprietary *implementation* (an example is Adobe's Postscript). While a competitive vendor is permitted to build a standard-compliant product, the dominant vendor hopes the development cost would be prohibitive and market acceptance would be minimal.

EXAMPLE: *Competing implementations have occurred, contrary to the wishes of the dominant supplier, in the PC BIOS (code embedded deeply within the bowels of a PC implementation). Originally designed by IBM, it was successfully reverse engineered and implemented by Phoenix Technologies. Another example is the Intel Pentium microprocessor, which has been cloned by Advanced Micro Devices and others. They created a design independently by replicating the functionality and specifications and without using the original design.*

De jure standards are rare in the computer industry, but occur in communications, which is often subject to government regulation (see Chapter 12). The regulatory process may dictate a single standard but leave it up to industry to determine its details.

A standard may be publicly documented but incorporate patented ideas and require adopters to pay royalties. In this case, companies contributing technology to a standard are able to retain patent rights but are obligated by the standards organization to freely license the technology to all comers for a "reasonable and nondiscriminatory" royalty.

Object Management Group

An important voluntary cooperative standardization effort is the Object Management Group (OMG) focused on object-oriented systems (see Chapter 6) and enterprise computing (see Section 2.6.2 on page 56). The OMG includes over 700 software companies who have found it in their best interest to join in promoting cross-platform standards, so that their products can participate together in enterprise applications. Strictly speaking, the OMG is not a standardization body, but rather simply makes recommendations as to the best technologies. Thus, it views its charter as the cooperative promulgation of de facto standards. The process followed by OMG is to identify areas that need standardization, request participating companies to contribute, evaluate those contributions by a technical committee, and make a final recommendation. Occasionally they ask members to merge their best ideas into a single proposal.

Further Reading

An extensive discussion of architecture design and the processes involved in creating them can be found in [BCK98]. A much more technical introduction to the design of distributed systems is [CDK94]. The business process reengineering described in [Dav93] is not dissimilar to the architecture design for software systems. Finally, strategies for using standardization as a competitive tool are discussed in [Sha98].