THIS PDF FILE
FOR PROMOTIONAL USE ONLY

7

Software Creation Industry

Competition has been shown to be useful up to a certain point and no further, but cooperation . . . begins where competition leaves off.

Franklin Delano Roosevelt

Chapter 6 addressed the industrial organization of the software value chain from software creation to use. Software creation, one of the more important links of the chain, starts with a set of requirements and culminates in a software distribution that can be provisioned, including analysis of user needs, development, maintenance, customer support, and upgrades (see section 5.1).

This chapter addresses the internal structure of the software creation industry. It is common for a total solution to be composed of software products from multiple firms, whether the integration is performed by a system integrator or by a software supplier who licenses modules from other suppliers. Thus, cooperation (as well as competition) among firms in the software industry is crucially important. Interesting issues addressed in this chapter include how the organization of this production industry arises, how and why it is changing, and how firms coordinate themselves to arrive at composable solutions meeting user needs. This chapter first discusses the industrial organization, then the ways in which firms coordinate themselves, and finally the supply chain arising within the industry based on software components.

7.1 Industrial Organization of the Software Industry

A relation between software architecture and industrial organization was pointed out in section 6.1; industry responsibility must follow interfaces of software modules at the top level of architectural decomposition. Is the architecture determined by the marketplace, or is industrial organization determined by architecture? What is

the industrial organization of the software creation industry, and how is it changing, and why? These issues are dealt with in this chapter.

7.1.1 Applications and Infrastructure

The most fundamental architectural concept in software is the decomposition into application and infrastructure. With some notable exceptions, firms in the industry generally specialize in one or the other.

Example There are three major types of exceptions. One is firms that combine the businesses of infrastructure supplier and application software supplier, for instance, Apple (particularly historically) and Microsoft. They strongly encourage independent application software suppliers to use their platforms but also supply their own applications. Another exception is firms that combine the business of infrastructure software supply and consulting services, the latter focused on helping end-user organizations acquire and provision new applications. Examples are IBM and Compaq, the latter a merger (Compaq and Digital) specifically designed to combine these businesses. A third exception is firms that combine infrastructure software with contract development of applications, for instance, IBM.

While both applications and infrastructure require technical development skills, the core competencies are different. Applications focus the value proposition on end-users, and infrastructure provides value primarily to application developers and to operators. Applications are valued most of all for functionality and usability; their performance and technical characteristics are more dependent on infrastructure. It is advantageous to move as much technical capability to the infrastructure as possible, so application developers can focus on user needs. This leads to a natural maturation process whereby novel technical functionality is first pioneered in leading-edge applications and then migrates to enrich infrastructure, or applications evolve (at least partially) into infrastructural platforms for other applications.

Example The playing of audio and video media was originally built into specialized applications. Today much of the required support is found in common infrastructure, usually at the level of operating systems. Individual productivity applications (such as office suites) have been augmented with ever richer user programmability support, so many interesting specialized applications now build on them (see section 4.2.7).

From a business perspective, application software has the advantage over infrastructure of providing value directly to the end-user, who ultimately pays for everything in the software value chain. This direct relationship provides rich opportunities

to differentiate from competitors and to leverage it for selling complementary products. Ceding this valuable direct relationship between supplier and user is a disadvantage of the application service provider model (from the supplier perspective) and also a motivation to become a service provider as well as a software supplier.

In some cases there are considerable marketplace obstacles to application adoption that make business difficult for application suppliers, such as lock-in and network effects (see chapter 9), but in other cases these are less important. Application software suppliers who provide variations on competitive applications find lock-in a greater obstacle but also benefit from a moderation of network effects, for instance, through providing backward compatibility or translations (Shapiro and Varian 1999b). There are many opportunities to pursue entirely new application categories, as illustrated by the recent explosion of Internet e-commerce.

As observed in section 3.1, applications are becoming increasingly numerous, diverse, and specialized. This is especially true of sociotechnical applications, which are often specific to the group or organizational mission they serve. This has several implications for industrial organization. First, a strong separation of applications and infrastructure reduces the barriers to entry to new application ideas. Where applications and infrastructure are supplied by separate firms, the latter find it advantageous to define open and well-documented applicator programming interfaces (APIs) that make it easier to develop applications, which in turn attracts application ideas from more sources and provides more diversity and competitive options to users. A good test of the application infrastructure separation is whether an application can be developed and deployed without the knowledge or cooperation of the infrastructure supplier or operator.

Second, application diversity is enhanced by doing whatever is necessary to make it faster, cheaper, and requiring less development skill. This includes incorporating more needed functionality in the infrastructure. Making use of software components (see section 7.3), and rapid prototyping and end-user programming methodologies (see section 4.2).

Third, application diversity is enhanced by an experimental approach seeking inexpensive ways to try out and refine new application ideas (see section 3.1.6). Applications should be a target for industrial and academic research, because a research environment is well suited to low-cost experiments and the refinement of ideas unfettered by the immediate goal of a commercially viable product (NRC 2000b) (see chapter 8). In reality, applications have traditionally not been an emphasis of the information technology (IT) research community for many reasons, including the importance of nontechnical considerations, the need for specific end-user

domain knowledge, the difficulty of gaining access to users for experimentation, and the inherent difficulty in assessing experimental outcomes.

Fourth, innovative new applications are a good target for venture capital funding and startup companies. The funding of competing startups is a good mechanism for the market to explore alternative application approaches. Venture capitalists specialize in managing the high risks of new applications and have effective mechanisms to abandon as well as start new businesses. This should not rule out large company initiatives, but large companies are usually not attracted by the limited revenue potential of a specialized application, put off by the financial risks involved, and sensitive to the opportunity costs of tying up scarce development resources.

Returning to the separation of application and infrastructure, the successes here also build on the economics underlying infrastructure (see chapter 9). If a new application requires a new infrastructure, then the required investment (as well as investment risk) is much larger than if the application is built on existing infrastructure. Thus, the separation of applications from infrastructure reduces barriers to entry and encourages small companies.

Example The history of the telephone industry illustrates these factors. Telephone companies are historically application service providers with one primary application—telephony. They are also infrastructure service providers, providing not only the infrastructure supporting telephony but also data communications (e.g., the Internet) and video (e.g., broadcast television distribution). They have shown interest in expanding their application offerings, primarily in directions with mass market appeal. In the United States, the telephone industry has launched three major application initiatives of this character: video telephony (extending telephony to include video), videotext (an early proprietary version of the Web), and video-on-demand. In all three cases, the financial risk in deploying an expensive capital infrastructure to support a new application with uncertain market potential proved too great, and the efforts were abandoned. The telephone industry also illustrates numerous successes in deploying applications building on the existing telephony infrastructure, including products from independent suppliers like the facsimile machine and voiceband data modem.

The telecommunications industry strategy addresses one serious challenge following from the complementarity of applications and infrastructure and from indirect network effects: an infrastructure supporting a diversity of available applications offers more value to users, and an application utilizing a widely

available infrastructure enjoys an inherently larger market. Industry thus faces the chicken-and-egg conundrum that a new infrastructure cannot be marketed without supported applications, and an application without a supporting infrastructure has no market. The telephone industry strategy has been to define a compelling application with mass market appeal and then to coordinate the investment in application and infrastructure, while making the infrastructure fairly specialized to support that application.[1]

The computer industry has generally followed the different strategy of deploying a generic infrastructure that supports a diversity of applications. In part this can be attributed to the culture of the industry, flowing from the original idea of programmable equipment whose application functionality is not determined at the time of manufacture. The Internet (a computer industry contribution to communication) followed a similar strategy; the core design philosophy for Internet technologies always valued low barriers to entry for new applications and a diversity of applications.

However, a pure strategy of deploying a generic infrastructure and waiting for applications to arrive is flawed because it does not address the issue of how to get infrastructure into the hands of enough users to create a market for applications that build on that infrastructure. The computer industry has found numerous ways to deal with this challenge (and has also suffered notable setbacks), all focused on making one or more compelling applications available to justify investment in infrastructure. An approach for totally new infrastructure is to initially bundle a set of applications with it, even while keeping the infrastructure generic and encouraging other application suppliers (e.g., the IBM PC and the Apple Macintosh were both bundled initially with a set of applications, and the Internet initially offered file transfer and e-mail). For infrastructure that has similar functionality to existing infrastructure, interoperability with older applications and offering higher performance characteristics for those older applications is another approach (e.g., layering; see section 7.1.3). Related to this, it is common for infrastructure to be incrementally expanded while maintaining backward compatibility for older applications. Application and infrastructure suppliers can explicitly coordinate themselves (e.g., by sharing product road maps; see section 7.2). Yet another approach is for applications to evolve into infrastructure by offering APIs or open internal interfaces (e.g., the Web; see section 7.1.2).

Another difference between the computer and telecommunications industries is the long-standing role of a service provider in telecommunications. Selling applications as a service bundled with a supporting infrastructure is advantageous in

providing a single integrated solution to customers and freeing them of responsibility for provisioning and operation. The software industry is moving in this direction with the application service provider model.

The goal should be to combine the most desirable features of these models, and indeed the separation of application and infrastructure at the technological level is not inconsistent with a service provider model and a bundling of application and infrastructure as sold to the user. One of the trade-offs involved in these strategies is summarized in the fundamental relationship (Shapiro and Varian 1999b).

Revenue = Market share × Market size.

An infrastructure that encourages and supports a diversity of applications exchanges market share (by ceding many applications to other suppliers or service providers) for an increase in total market size (by providing more diversity and value to users). Just as software suppliers must decide on their degree of application/infrastructure separation, service providers face similar issues. They can offer only applications bundled with infrastructure, or enhance the diversity of application offerings while ceding revenues and part of the customer relationship by giving third-party application providers access to their infrastructure. To maximize revenues in the latter case, use-based infrastructure pricing models can maximize the financial return from application diversity. These issues will become more prominent with the emerging Web services (see section 7.3).

7.1.2 Expanding Infrastructure

The growing cost of software development and the shortage of programming professionals concerns software development organizations. This is exacerbated by the increasing specialization and diversity of applications (see section 3.1.5); specialized applications may be economically feasible only if development costs can be contained. Several trends reduce developments costs, including improved tools, rapid development methodologies (see section 4.2), greater use of software components and frameworks (see section 7.3.6), and expanding infrastructure to make it cheaper and faster to develop and deploy applications.

The idea behind expanding infrastructure is to observe what kind of functionalities application developers reimplement over and over, and to capture those functionalities in a generic and flexible way within the infrastructure. It is important to capture these capabilities in a generic and general way so that they can meet the needs of a wide range of present and future applications. End-users for infrastructure software include application developers and operators.

Example Many applications need authentication and access control for the end-user (see section 5.4). Many aspects of this capability are generic and separated from the specific needs of each application. If authentication and access control are included within the infrastructure to be invoked by each application for its own purposes, reimplementation is avoided and users benefit directly by being authenticated only once for access to multiple applications.

These economic realities create an opportunity for the infrastructure to expand in capability over time. This may happen directly, or sometimes software developed as part of an application can be made available to other software and subsequently serve as infrastructure.

Example Early applications had to manage much of the graphical user interface on their own, but later this capability was moved to the operating system (initially in the Apple Macintosh). Software to format screen documents based on the Web markup language (HTML) was first developed in the Web browser but was also potentially useful to other applications (like e-mail, which frequently uses HTML to format message bodies). For example, Microsoft made this HTML display formatting available to other applications in its Windows operating system and to the system itself in displaying help screens. The company provided an API to HTML formatting within the Internet Explorer browser and included the Internet Explorer in the Windows software distribution.

Sometimes, an entire application that becomes ubiquitous and is frequently composed into other applications effectively moves into the infrastructure category.

Example The Web was originally conceived as an information access application for scholarly communities (World Wide Web Consortium 2002) but has evolved into an infrastructure supporting e-commerce and other applications. Many new distributed applications today incorporate the Web server and browser to present application-specific information to the user without requiring application-specific client software. Office suites are commonly used as a basis for custom applications serving vertical industry markets.

Valued-added infrastructure adds additional capability to an existing infrastructure.

Example A major area for innovation in infrastructure is *middleware*, defined roughly as infrastructure software that builds on and adds value to the existing network and operating system services. Middleware sits between the existing infrastructure and applications, calling upon existing infrastructure services to provide enhanced or extended services to applications. An example is *message-oriented*

middleware, which adds numerous message queuing and prioritization services valuable to work flow applications.

Market forces encourage these additions because of the smaller incremental investments compared to starting anew and because of the ability to support legacy applications utilizing the earlier infrastructure. From a longer-term perspective, this is problematic in that it tends to set in stone decisions made earlier and to introduce unnecessary limitations, unless designers are unusually visionary. Economists call these *path-dependent effects*.

Example Early Internet research did not anticipate streaming audio and video services. The core Internet infrastructure therefore does not include mechanisms to ensure bounded delay for transported packets, a capability that would be useful for delay-sensitive applications like telephony or video conferencing.[2] While acceptable quality can be achieved without these delay guarantees, better quality could be achieved with them. Unfortunately, once a packet is delayed too much, there is no way to make up for this, as time moves in only one direction. Hence, no value-added infrastructure built on the existing Internet technologies can offer delay guarantees—a modification to the existing infrastructure is required. Value-added infrastructure lacks complete freedom to overcome earlier design choices, particularly in performance dimensions.

The chicken-and-egg conundrum—which comes first, the applications or the infrastructure they depend on—is a significant obstacle to establishing new infrastructure capability. One successful strategy has been to move infrastructure with a compelling suite of applications into the market simultaneously, presuming that even more applications will come later.

Example The Internet illustrates this, as it benefited from a couple of decades of refinement in the academic research community before commercialization. A key was developing and refining a suite of "killer apps" (e.g., file transfer, e-mail, Web browsing). This, plus an established substantial community of users, allowed the Internet to reach commercial viability and success quickly once it was made commercially available. This is an oft-cited example of the important role of government-funded research (see chapter 8), subsidizing experimentation and refinement of infrastructure and allowing a suite of compelling applications to be developed. Such externally funded experimental infrastructure is called a test bed for the new space to be populated.

Middleware illustrates another strategy. Applications and (future) infrastructure can be developed and sold as a bundle while maintaining strict modularity so that

the infrastructure can later be unbundled and sold separately. A variation is to establish APIs to allow independent use of capabilities within an application.

Example A way to establish a message-oriented middleware (MOM) product might be to develop and bundle it with an enterprise work flow application, such as a purchase order and accounts payable application. By providing open APIs to the MOM capabilities, other application suppliers are encouraged to add application enhancements or new application capabilities that depend on the MOM. If this strategy is successful, eventually the MOM assumes a life of its own and can be unbundled and sold separately as infrastructure.

7.1.3 Vertical Heterogeneity: Layering

The modularity of infrastructure is changing in fundamental ways, driven primarily by the convergence of the computing (processing and storage) and telecommunications industries. By *convergence*, we mean two industries that were formerly independent becoming competitive, complementary, or both. This convergence is manifested primarily by the Internet's enabling of globally distributed software (see section 4.5), leading to applications that emphasize communication using distributed software (see section 3.1). This led to competing data networking solutions from the telecommunications and computer industries[3] and made networking complementary to processing and storage.

The top-level vertical architecture of both the telecommunications and computer industries prior to this convergence resembled a *stovepipe* (see figure 7.1). This architecture is based on market segmentation, defining different platforms for different application regimes. In the case of computing, mainframes, servers (originally minicomputers and later microprocessor-based) and desktop computers were introduced into distinct market segments (see table 2.3), each segment offering typically two or three major competitive platforms. Each segment and platform within that segment formed a separate marketplace, with its own applications and customers. Mainframes served back-office functions like accounting and payroll, servers supported client-server departmental functions like customer service, and desktop computers served individual productivity applications.

Similarly, the telecommunications industry segmented the market by application or information medium into telephony, video, and data. Each of these media was viewed as a largely independent marketplace, with mostly separate infrastructure sharing some common facilities.

Example Telecommunications firms have always shared right-of-way for different applications and media, and also defined a digital multiplexing hierarchy (a recent

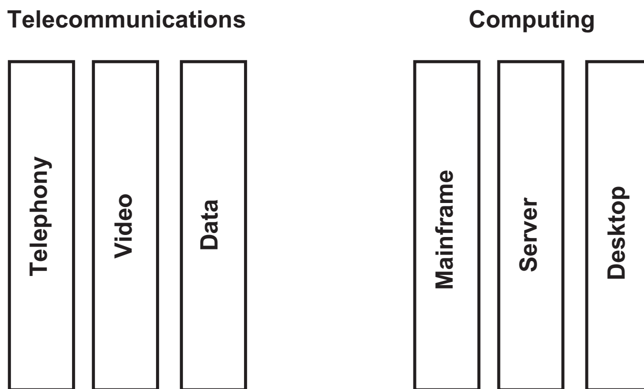

Figure 7.1
Historically, the telecommunications and computing industry both used an architecture resembling a stovepipe.

example is SONET, or synchronous optical network) that supported a mixture of voice, data, and video services.

While the telecommunications and computer architectures look superficially similar, historically the approach has been different, primarily arising out of the importance of the service provider in telecommunications but not in computing. With notable exceptions, in telecommunications the infrastructure and application suppliers sold to service providers, who did the provisioning and operation, and the service providers sold application services (and occasionally infrastructure services) to users. In computing, it was common for infrastructure suppliers to sell directly to users or end-user organizations, who acquire (or develop themselves) applications and do their own provisioning and operation. This is partly due to the different cultures and the relative weakness of data networking technologies (necessary to sell application services based on processing and storage) in the early phases of the computer industry.

These distinct industry structures led to fundamental differences in business models. Firms in the telecommunications industry historically saw themselves as application service providers, viewed the application (like telephony or television-video distribution) as their business opportunity, and constructed a dedicated infrastructure for their application offerings. Infrastructure was a necessary cost of business to support applications, the primary business opportunity. Further, service providers viewed alternative applications and application suppliers as a competitive threat.

In contrast, the relatively minor role of a service provider in the computer industry and the cultural influence of the technical genesis of computing (programmability, and the separation of application from infrastructure) resulted in a strikingly different business model. Infrastructure and application suppliers sold independently to end-user organizations, and the users integrated the two. As a result, neither the application supplier nor the user perceived much freedom to define new infrastructure but focused on exploiting existing infrastructure technologies and products. The infrastructure supplier encouraged a diversity of complementary applications and application suppliers to increase the value of its infrastructure and simultaneously provide customers better price, quality, and performance through application competition.

To summarize the difference in the telecommunications and computing business strategies, in telecommunications the infrastructure chased the applications, whereas in computing the applications chased the infrastructure. While there are many necessary qualifications and notable exceptions to this statement, for the most part it rings true. In a sense, the telecommunications business model formed a clearer path to dealing with the indirect chicken-and-egg network effects mentioned earlier. Regardless of whether applications chase infrastructure or the reverse, investments in new infrastructure technologies have to proceed on faith that there will be successful applications to exploit new infrastructure. In the telecommunications industry this was accomplished by directly coordinated investments, and in the computer industry an initial suite of applications was viewed as the cost of establishing a new infrastructure.

Example To complement its PC, IBM initially supplied a suite of personal productivity applications, as did Apple Computer with the Macintosh. In both cases, open APIs in the operating system encouraged outside application developers, and it was not long before other application software suppliers supplanted the original infrastructure supplier's offerings (particularly for the PC).

This is all historical perspective, and not an accurate view of the situation today, in part because of the convergence of these two industries. The infrastructure has shifted away from a stovepipe form and toward a horizontal architecture called *layering*. The layered architecture organizes functionality as horizontal layers (see figure 7.2), each layer elaborating or specializing the functionality of the layer below. Each layer focuses on supporting a broad class of applications and users rather than attempting to segment the market. A natural way to enhance and extend the infrastructure is to add a new layer on top of existing layers. If applications are permitted to directly access services from lower layers, the addition of a new layer does

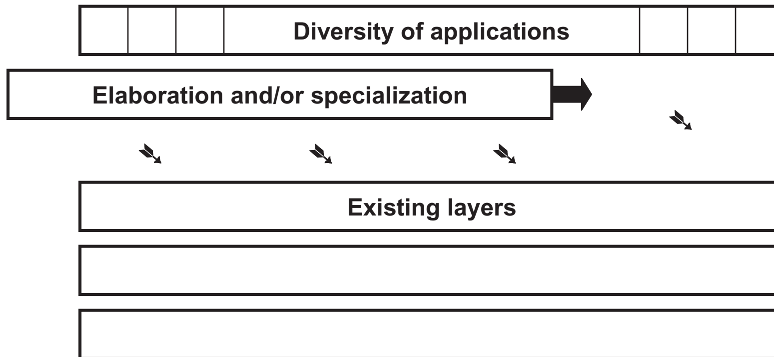

Figure 7.2
The layered architecture for infrastructure modularizes it into homogeneous horizontal layers.

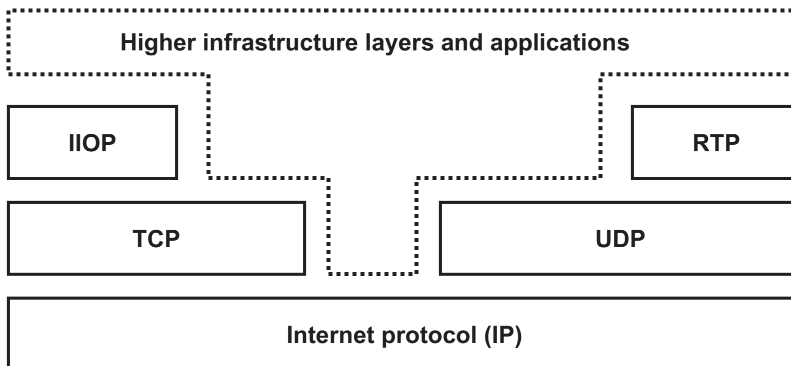

Figure 7.3
A simplified architecture of the Internet illustrates how new layers are added while future layers and applications can still invoke services of previous layers.

not disrupt existing applications but creates opportunities for new applications. While applications are allowed to access any layer, each layer is usually restricted to invoke only the services of the layer immediately below. This restriction can be eased by allowing two or more layers in parallel, at the same level.

Example As illustrated in figure 7.3, the Internet is built on a foundation layer called the internet protocol (IP) and on top of that two widely used layers, transmission control protocol (TCP) and user datagram protocol (UDP). IP offers a service that conveys packets from one host to another with no guarantee of delivery order or reliability (analogous to sending postcards through the postal system).

TCP and UDP invoke the services of IP to direct packets to a specific application running on the host. TCP also offers reliability and guaranteed ordering of delivery, achieved by detecting lost (or excessively delayed) packets and resending them. Later, the internet inter-ORB protocol (IIOP) and the real-time protocol (RTP) layers were added on top of TCP and UDP, respectively, to support distributed object-oriented applications and streaming audio and video. HTTP (hypertext transfer protocol) is an important protocol layered on TCP, the main protocol underlying the Web and easily recognized in Web addresses (http://). In the future, more layers may be added; future layers and applications are permitted to access lower layers. Applications can even invoke IP services directly, although this would be unusual.

Since the layered architecture dominates the converged computing and telecommunications industries, it largely displaces the stovepipe architecture historically characteristic of both industries. There are several forces driving this shift and accompanying implications for industrial structure. The first is the effect of the Internet on the computer industry. By enabling distributed applications to communicate across platforms, it creates a new technical and commercial complementarity among them. End-users do not want to uniformly adopt a single platform to use a particular distributed application, nor do they want to participate in an application with only a subset of other users, reducing value because of network effects. Suppliers of new distributed applications don't want to limit their market to a subset of users on a given platform, or take specific measures to support different platforms. For applications to be easy to develop, deploy, and operate in an environment with heterogeneous platforms, application suppliers want to see homogeneity across those platforms. Horizontal homogeneity can potentially be achieved in a layered architecture by adding layers above the existing heterogeneous platforms, those new layers hiding the heterogeneity below. Because of path-dependent effects, this results in a hybrid stovepipe-layered architecture.

Example The virtual machine and associated environment for code portability can be viewed as a layer added to the operating system within each platform (see section 4.4.3). As illustrated in figure 7.4, this new layer adds a uniform execution model and environment for distributed software applications. It can be viewed as a homogeneous layer that sits on top of existing heterogeneous platforms (like Windows, Mac OS, and different forms of UNIX). Further, there is no reason (other than inconvenience in provisioning and application composability) not to have two or more parallel virtual machine layers supporting different groups of applications.

A layer that hides the horizontal heterogeneity of the infrastructure below and is widely deployed and available to applications is called a *spanning layer*. The most

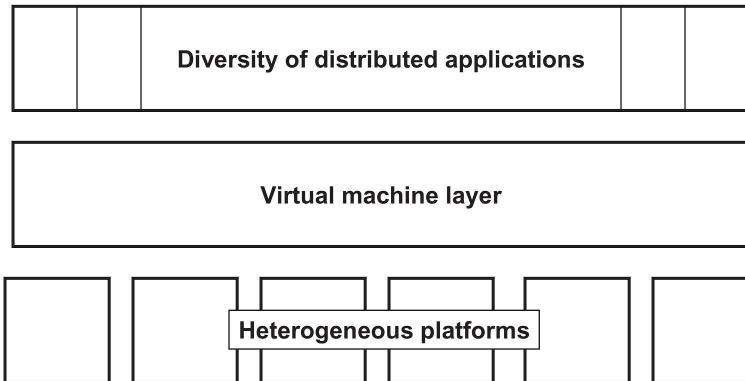

Figure 7.4
The widely deployed virtual machine can create a homogeneous spanning layer for applications that hides the heterogeneity of platforms.

important spanning layer today, the internet protocol, was specifically designed to hide heterogeneous networking technologies below. The virtual machine is another example of a spanning layer, arguably not yet widespread enough to deserve this appellation. One way to view the relation between these spanning layers was illustrated earlier in the "hourglass" of figure 4.6.

A second driver for layering is the trend toward applications that integrate processing, storage, and communication and mix data, audio, and video (see section 3.1). In contrast to the stovepipe, each horizontal layer (and indeed the entire infrastructure) supports a variety of technologies, applications, and media.

Example Within the communication infrastructure, the IP layer has been extended to support data by the addition of the TCP layer and extended to support streaming audio media by the addition of an RTP layer on top of UDP (see figure 7.3). A given application can mix these media by accessing the TCP layer for data and the RTP layer for audio and video.

A third and related driver for layering is value added to infrastructure that can support the composability of different applications (see section 3.2.12), which is one of the most important roles of infrastructure. By binding different applications to different infrastructures, a stovepipe architecture is inherently constrained in its ability to support composability, but a layered architecture is not.

A fourth driver for layering is its allowance for incremental extension and elaboration while continuing to support existing applications. This reduces the barrier to entry for applications that require new infrastructure capabilities, since

most of the supporting infrastructure does not need to be acquired or provisioned. Looking at it from the computer industry perspective (infrastructure first, applications later), this allows incremental investments in infrastructure for both supplier and customer.

Modularity introduces inefficiency, and layering is no exception. Compared to a monolithic stovepipe, layering tends to add overhead, no small matter in a shared infrastructure where performance and cost are often important. The processes described by Moore's law are thus an important enabler for layering (see section 2.3).

Layering fundamentally shifts the structure of industry competition. Each layer depends on the layers below (they are complementary), and an application requires the provisioning and operation of all layers upon which it depends. This creates complementary infrastructure suppliers, and a burden on infrastructure provisioning to integrate layers. Competition in the infrastructure is no longer focused on segmentation of the market for applications but rather on competition at each layer, each supplier attempting to provide capabilities at that layer for a wide range of applications. The integration of layers requires coordination among suppliers (see section 7.2), and functionality and interfaces are fairly constrained if alternative suppliers are to be accommodated.

Layering fundamentally changes the core expertise of industry players. Expertise about particular application segments no longer resides with infrastructure suppliers but primarily within application suppliers. Market forces encourage infrastructure suppliers to extend the capabilities they supply to serve all (or at least a broader range of) applications because this increases market size. This increases their needed range of expertise, and if this proves too daunting, they may narrow their focus vertically by specializing in only one or two layers. Startup companies especially face a challenge in this industry structure because of the generality and hence high development costs and wide-ranging expertise required. Thus, startup companies tend to focus either at the top (applications) or bottom (technology) of the layering architecture, where diverse solutions thrive and there are fewer constraints and less need to coordinate with others (see section 7.1.5.).

Example The physical layer of communication (transporting a stream of bits via a communication link) is a ripe area for startup companies, especially in light of the variety of media available (optical fiber, radio, free-space optical, coaxial cable, and wirepair). As long as they interface to standard solutions for the layers above, innovation is relatively unconstrained. The analogous opportunity in processing is the microprocessor, so one might expect a similar diversity of startups. In fact, microprocessor startups are rare because the instruction set is deeply intertwined

with the software layers above, greatly limiting the opportunity for differentiation. The emulation or virtual machine idea is one way to address this, but this reduces performance, one of the prime means of differentiation. An interesting attempt at combining the virtual machine and the custom microprocessor concepts is Transmeta's Crusoe, a small, energy-efficient processor with a proprietary instruction set complemented by a software layer that translates standard Intel instruction sequences into Crusoe instructions.

It is useful to examine the appropriate modularity of layering in more detail. It is striking that no single organization has responsibility for consciously designing the overall layered architecture. Rather, it is determined by research and company initiatives, collaborations among companies, and standardization bodies. The result is "creative chaos" that introduces strengths and weaknesses. On the plus side, innovations are welcome from many quarters, and good ideas have a reasonable chance of affecting the industry. On the negative side, application suppliers and provisioness must deal with a lot of uncertainty, with competing approaches to consider and no clear indication as to which ones will be successful in the long term.

Example The first successful attempt at enabling cross-platform middleware as a spanning layer was the Object Management Group's common object request broker architecture (CORBA), a suite of standards to enable distributed object-oriented applications. CORBA has been successful in intraenterprise integration, where platform variety arises out of acquisitions and mergers and yet cross-platform integration is required. CORBA did not achieve similar success in interenterprise integration, where heterogeneous platforms are even more prevalent. A later approach to cross-platform integration was Java, now usually used in conjunction with CORBA in enterprise solutions. Again, interenterprise integration remains beyond reach for technical reasons. The latest attempt at global integration is Web services based on XML (extended markup language) and other Web standards (see section 7.3.7). With Web services emerging as the most likely universal spanning layer, competition in the layer immediately below heats up: Microsoft's. NET Common Language Runtime and its support for Web services compete against the Java virtual machine and its emerging support for Web services.

Historically, the approach was very different in the telecommunications industry. This arguably resulted in less change and innovation (but still a remarkable amount) but in a more reliable and stable infrastructure.

Example Until about two decades ago, each country had a monopoly national telephone service provider (often combined with the post office). In the United States

this was the Bell System, with its own research, equipment, software development, and manufacturing. Suppliers and providers coordinated through standardization bodies in Geneva, predominantly the International Telecommunication Union (ITU), formerly called Comité Consultatif International Téléphonique et Télégraphique (CCITT). Through these processes, the national networks and their interconnection were carefully planned top-down, and changes (such as direct distance dialing) were carefully planned, staged, and coordinated. This resulted in greater reliability and stability but also fewer competitive options or diversity of choice.

Since the networked computing infrastructure has not followed a top-down process, beyond the core idea of layering there is no overall architectural vision that guides the industry. Rather than pointing to a guiding architecture, we must resort to an analysis of the current state of the industry. An attempt at this analysis is shown in figure 7.5 (Messerschmitt 1999b). It illustrates three stovepipes of lower layers, one specific to each technology (processing, storage, and connectivity). Distributed applications (as well as nondistributed applications that combine processing and mass storage) want a homogeneous infrastructure that combines these three technologies in different ways, and thus the top layers are common to all three (see table 7.1).

The essential idea behind figure 7.5 is illustrated in figure 7.6. The intermediate layers provide a common set of services and information representations widely used

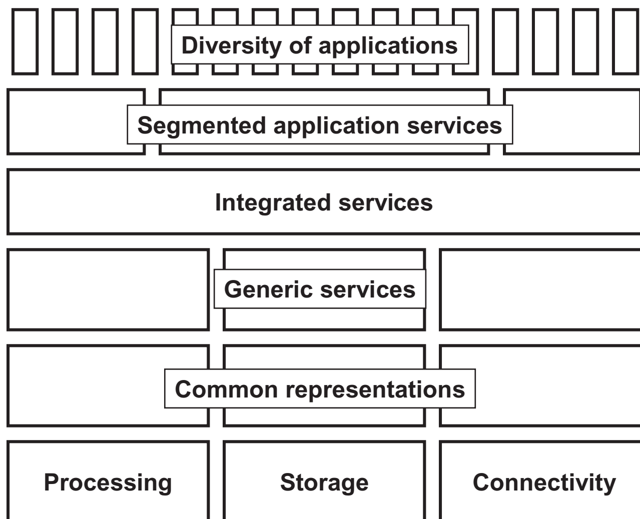

Figure 7.5
A layered architecture for distributed applications and the supporting infrastructure.

Table 7.1
Description of Layers Shown in Figure 7.5

Layer	Description	Examples
Applications	A diversity of applications provide direct and specific functionality to users.	
Segmented application services	Captures functionality useful to a narrower group of applications, so that those functions need not be reimplemented for each application. This layer has horizontal heterogeneity because each value-added infrastructure element is not intended to serve all applications.	Message-oriented middleware emphasizes work flow applications; information brokering serves as an intermediary between applications or users and a variety of information sources.
Integrated services layer	Provides capabilities that integrate the functions of processing, storage, and connectivity for the benefit of applications.	Directory services use stored information to capture and identify the location of various entities—essential to virtually all distributed applications.
Generic services layer	Provides services that integrate processing, storage, and connectivity in different ways.	The reliable and ordered delivery of data (connectivity); the structured storage and retrieval of data (storage); and the execution of a program in an environment including a user interface (processing and display).
Common representations layer	Provides abstract representations for information in different media (like processing instructions, numerical data, text, pictures, audio, and video) for purposes of processing, storage, and communication. They are deliberately separated from specific technologies and can be implemented on a variety of underlying technologies.	A virtual machine representing an abstract processing engine (even across different microprocessors and operating systems); a relational table representing the structure of stored data (even across different platforms); and a stream of bytes (eight-bit data) that are delivered reliably in order (even across different networking technologies).
Processing, storage, and connectivity	Provide the core underlying technology-dependent services.	Microprocessors, disk drives, and local-area networking.

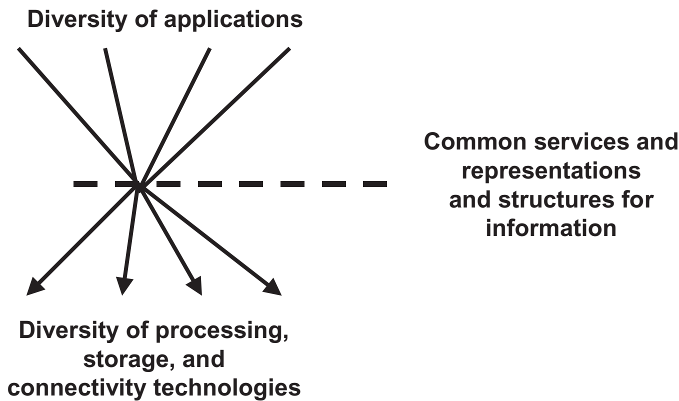

Figure 7.6
Layering provides separation of a diversity of technologies from a diversity of applications.

by applications. The goal is to allow a diversity of technologies to coexist with a diversity of applications without imposing the resulting chaos on applications—applications and technologies can evolve independently without much effect on each other. Providing a reimplementation of the common representation and services layers for each distinct technology accommodates this.

Of particular importance is the spanning layer. Assuming it is not bypassed—all layers above make use of its services but do not interact directly with layers below—a well-designed spanning layer can eliminate the dependence of layers above from layers below, allowing each to evolve independently. Successfully establishing a spanning layer creates a large market for solutions (application and infrastructure) that build upon it, both above and below. The spanning layer brings to bear the positive feedback of network effects without stifling technical or competitive diversity of layers below. It illustrates the desirability of separating not only application from infrastructure but also infrastructure from infrastructure.

Example The internet protocol can be viewed as a spanning layer, although it is limited to the connectivity stovepipe. As illustrated by the hourglass of figure 4.6, the IP layer does effectively separate applications from underlying networking technologies and has become virtually ubiquitous. Suppliers creating new communication and networking technologies assume they must support an IP layer above, and application suppliers assume they can rely on IP layers below for connectivity. Applications need not be redesigned or reconfigured when a different networking technology (e.g., Ethernet local-area network, wireless local-area network, fiber-optic

wide-area network, wireless wide-area network, or satellite network) is substituted. The existence of IP also creates a ready and large market for middleware products building on internet protocols.

There are, however, limitations to layering. Mentioned earlier is the added overhead necessary to implement any strong modularity, including layering. In addition, intermediate layers can hide functionality but not performance characteristics of the underlying technology from applications.

Example When the user substitutes a slow network access link for a faster one, the delay in packet delivery due to transmission time will be increased. Nothing in the intermediate layers can reverse this.

The preferred approach today to dealing with performance variations is to make applications robust and adaptive to the actual performance characteristics.[4] Applications should be able to take advantage of higher-performance infrastructure and offer the best quality they can subject to infrastructure limitations.

Example A Web browser-server combination will display requested pages with low delay when there is ample processing power and communication bandwidth. When the bandwidth is much lower (say a voiceband data modem), the browser and server should adjust by trading off functionality and resolution for added delay in a perceptually pleasing way. For example, the browser may stop displaying high-resolution graphics, or ask the server to send those graphics at a lower resolution, because the resulting diminution in delay more than compensates perceptually for lost resolution.

Many current industry standardization and commercialization efforts would support the layered model (see figure 7.7 for examples). For each standard illustrated, there are standards competing for adoption. At the common representation layer, the Java virtual machine, the relational table, and the Internet's TCP are widely adopted. At the generic services layer are shown three standards that support object-oriented programming (OOP), a standard programming technique that emphasizes and supports modularity (see section 4.3). Programs constructed according to this model consist of interacting modules called objects, and the generic services layer can support execution, storage, and communication among objects. Java supports their execution, the object-relational database management system (ORDBMS) supports the storage of objects, and IIOP allows objects to interact over the network in much the same way as they would interact within a single host.

At the integrative services layer, CORBA attempts to identify and standardize a set of common services that integrate processing and connectivity (by incorporat-

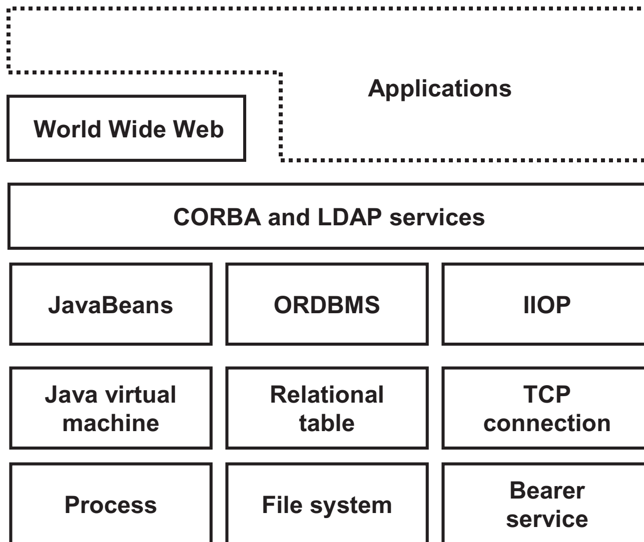

Figure 7.7
Examples of industry standards fitting the layered model of figure 7.5.

ing Java mobile code capabilities) and processing and storage (by providing for the storage of objects). Examples include the creation or storage of objects on demand, and directories that discover and locate objects and services on the network to make distributed applications easier to develop. The Web was mentioned earlier as an application that "grew up" to become an infrastructure supporting applications that access and update information over the network. On the other hand, the Web does not support all applications (e.g., those not based on a client-server architecture). Thus, the Web-as-infrastructure falls at the segmented application services layer.

7.1.4 Core Competencies

Earlier, the historical approaches of the telecommunications and computer industries were contrasted. This contrast raises an important issue for industrial organization: Who is responsible for provisioning and operation? As described in section 6.2, there are three primary options: an independent service provider (the historical telecommunications industry model), the application or infrastructure supplier (rare), or the user (the historical computer industry model). Of course there are other options, such as splitting responsibility for provisioning and operation, application and infrastructure, or different parts of the infrastructure.

The increasing role of application service providers in the software industry, and the trend in the telecommunications industry to focus on the provisioning and operation of infrastructure and not applications (particularly in the Internet segment of their market) suggests that radical change in industry structure may be occurring. This change can be traced to at least three undercurrents. One is the ubiquity and performance of the Internet, which opens up the option of operations shifted to a service provider while making application functionality available over the wide-area network. From the user perspective it makes no difference where the operations reside, except for important factors like performance, availability, and customer service. A second undercurrent leading to organizational change is the growing specialization and diversity of applications, and the resulting importance of application suppliers' focusing their efforts on satisfying user needs and requirements. Infrastructure suppliers and service providers have not proven as effective at this as more specialized application suppliers; this suggests an organizational separation of applications and infrastructure.

These observations provide hints as to how the industrial organization may evolve in the future. Like good software architecture (see section 4.3), market forces encourage an industrial organization with weak coupling of functions and expertise across different companies and strong cohesion within companies. These properties can be interpreted in different ways, such as transaction costs and coupling of expertise. In the long term, it seems that market forces reward firms that specialize in certain responsibilities but share core competencies, because many managerial approaches, such as compensation and organizational structures and processes, are tied to these competencies. Of course, there are many other considerations of importance, such as the desire of customers for a complete product portfolio or turn-key solution, or opportunities to gain competitive advantage by synergies among complementary responsibilities.

This suggests a fresh look at the software value chain (see section 6.2), not in terms of responsibilities but in terms of core competencies. Seven core competencies can be identified (see table 7.2).

To the extent that industrial organization is presaged by natural groupings of core competencies, the independent service provider (as embodied in the application service provider model) seems a likely outcome, because the core competencies resident there are quite distinct from those of the other roles. Telecommunications service providers should not try to be exclusive application suppliers; rather, they should focus on making their infrastructure services suitable for a wide range of applications and encourage a diversity of applications from many sources. They may

Table 7.2
Core Competencies Relating to the Software Industry

Competency	Description
Business function	An end-user organization should understand its business functions, which in most cases are not directly related to software or technology or software-based services.
User needs and requirements	Industry consultants should understand end-user needs, which increasingly requires specialized knowledge of an industry segment or specific organizational needs, in order to help organizations revamp business models, organization, and processes to take maximum advantage of software technology.
Application needs	Infrastructure suppliers should understand needs common to a wide variety of applications and application developers, and also market forces that strongly influence the success or failure of new infrastructure solutions.
Software development	Both application and infrastructure software suppliers need software development and project management skills, with technical, organizational, and management challenges. Application suppliers must also be competent at human-centered considerations such as user interfaces.
Provisioning	Constrained by the built-in flexibility and configurability of the application, the system integrator and business consultant must understand unique organizational needs and be skilled at choosing and integrating software from different firms.
Operation	Operators must be competent at the technical aspects of achieving availability and security, administrative functions like trust and access, and customer service functions such as monitoring, billing, and helpdesk.

exploit their core competency in operation by extending it from today's narrow range of applications (telephony, video distribution) to a wider range acquired from independent application suppliers, increasing the diversity of application service providers' offerings.

The increasing diversity and specialization of applications, and the need to consider the organizational and process elements of information technology and the interface between organization and technology, have profound implications for application software suppliers. As the core competencies differ sharply from software development, these enterprise and commerce software suppliers should look more and more to industry consultants to assist in needs and requirements definition.

For end-user organizations, focusing on core competencies would imply outsourcing application development to application software suppliers, and provisioning and operation to system integrators, consultants, and service providers. In fact, this is becoming prevalent.

7.1.5 Horizontal Heterogeneity

From a technical perspective, an infrastructure layer that is horizontally homogeneous is advantageous (see section 7.1.3). This is an important property of a spanning layer because it creates a large market for layers above and below that can evolve independently. However, as shown in figure 7.5, it is entirely appropriate for horizontal heterogeneity to creep into the layers near the top and the bottom. Near the top, this segments the market for infrastructure to specialize in narrower classes of applications. With platforms supporting applications, one size does not fit all.

Example Distributed applications benefit from an infrastructure that hides the underlying host network structure, but this is unnecessary overhead for applications executing on a single host. Work flow applications benefit from a message and queuing infrastructure, and online transaction processing applications benefit from a database management system.

Near the bottom, it is desirable to support a diversity of technologies. This diversity arises out of, as well as encourages, technological innovation.

Example Innovation in microprocessor architecture has accompanied Moore's law as an important enabler for improving performance. Sometimes these architectural innovations can be accomplished without changing the instruction set—which clearly contributes to horizontal heterogeneity—but innovations in instruction sets enable greater advances. An example is the idea of a reduced instruction set computer (RISC), which traded simplicity in the instruction set for higher instruction execution rates. Numerous technological innovations have spawned heterogeneity in storage (e.g., recordable optical disks) and communication (e.g., wireless) as well. As underlying technology changes, so does the implementation of the lower layer infrastructure.

History and legacy technologies are another reason for heterogeneity.

Example Many historical operating system platforms remain, and several remain vibrant, evolving, and attracting new applications. The Internet brought with it distributed applications and network effects that place a premium on interoperability across platforms, e.g., a Web server running on a UNIX server and a Web browser running on a Macintosh desktop platform. The Internet has also brought with it a

need for composability of different applications running on different hosts. For example, MIME[5] is a standard that allows a wide variety of applications on different platforms to agree on content types and the underlying data formats and encodings (it originated to support e-mail attachments but is now used more broadly).

It was argued earlier that unconditional software portability is neither a practical nor a desirable goal (see section 4.4.3), and for the same reason evolving toward a single platform is not desirable. There is no predetermined definition of which functionalities belong in applications and which in infrastructure, but rather capabilities that many applications find valuable (or that keep appearing in multiple applications) work their way into the underlying platform. The commonality inherent in ever-expanding platforms enables greater diversity of applications and especially supports their interoperability and composability. That this can occur in multiple platforms speeds the diversity of ideas that can be explored, and offers application developers some choice among differentiated platforms. At the same time, industry must deal with the reality of heterogeneous platforms, especially for distributed applications that would otherwise become Balkanized, confined to one platform and a subset of users, with the resulting network effect diminution of value. Fortunately, the owners of individual platforms—especially those with a smaller market share and especially with the rising popularity of distributed applications—have a strong incentive to ensure that their platforms can participate in distributed applications with other platforms. But specifically what can they do about it? Adding a layer with the hope that it becomes widespread enough to be considered a spanning layer is one approach.

Example Sun's Java effort, including the Java programming language, runtime environments (including virtual machines), libraries, and interfacing standards, created such a candidate for spanning layer, abstracting from underlying platforms, software, and hardware. Other examples include the CORBA standards and Microsoft's COM and .NET.

However, given the difficulties of establishing a totally new layer that achieves a high enough market penetration to attract a significant number of applications, new layers can arise out of an existing application or infrastructure.

Example The Web migration from application to infrastructure resulted from two factors. First, it had become ubiquitous enough to attract application developers, who were not ceding much market potential by adopting it as a foundation. Second, the Web provided open APIs and documented internal interfaces that made it relatively straightforward to exploit as an infrastructure. This illustrates that an

application that is more open or less proprietary is more likely to be adopted as the basis for other applications, that is, as infrastructure.

Another response to heterogeneous platforms is to design an application to be relatively easy to port to different platforms. This places special requirements on interfaces between the application and its underlying platform; if they are designed in a relatively platform-independent and open way, the porting becomes easier (but exploiting the strengths of specific platforms becomes harder).

Example The Web is a good illustration of this. As a distributed application, the Web had to deal with two primary challenges. First, if the Web was to be more than simply a vehicle for displaying static stored pages (its original purpose), it had to allow other applications to display information via the Web browser and server. For example, dynamic Web pages based on volatile information stored in a database management system required an intermediary program (encompassing what is usually called the application logic) that requested the appropriate information from the database and displayed it in the proper formats in the browser. For this purpose, an open and relatively platform-independent API called the common gateway interchange (CGI) was standardized. The second challenge was the interoperability between browser and server running on different platforms. Fortunately, this problem was already partially solved by IP, which provided a communication spanning layer that allowed one host to communicate data to another using a standard packet representation, and a TCP that provided reliable and ordered transport of a stream of bytes. The Web simply had to standardize an application-layer transfer protocol (HTTP). These open standards made the Web relatively platform-independent, although different versions of the browser and server still had to be developed for the different platforms. Later, these open interfaces were an important factor in the evolution of the Web into infrastructure. For example, other applications could make use of HTTP to compose the entire browser or server into an application, or could even use HTTP independently (e.g., Groove, an application that uses HTTP as a foundation to share files and other information among collaborating users).[6] The latest generation of standardization focuses on Web services, and one of the core protocols in this space (SOAP: simple object access protocol) usually operates on top of HTTP (see section 7.3.7).

Another approach is to embrace horizontal heterogeneity but arrange the infrastructure so that interoperability and even composability are achieved across different platforms by appropriate industry standards or by platform-specific translators.

Example Different platforms for both servers and desktop computers tend to use different protocols and representations for file storage and print services. Users, on the other hand, would like uniform access to their files across all platforms (including Mac OS, the different forms of UNIX, and Windows), for example, to access files or print services on a UNIX server from a desktop (Linux or Mac OS or Windows) platform. Various open source solutions (e.g., Samba) and commercial solutions (e.g., Netware, NFS, Appletalk, Banyan Vines, and Decnet) provide this capability. See section 7.3 for additional examples in the context of software components and Web services.

Instead of tackling protocols (which specify how information gets from one platform to another) other industry standards focus on providing a common, flexible representation for information (without addressing how that information gets from one platform or application to another). These are complementary, since information must be transferred and must be understandable once it has been transferred.

Example XML is a flexible and extensible language for describing documents. It allows standardized tags that identify specific types of information to be identified. A particular industry can standardize these tags for its own context, and this allows different firms to exchange business documents and then extract the desired information from those documents automatically. For example, one industry might define an XML-based standard for describing purchase orders, and then each company can implement translators to and from this common representation for purchase orders within its (otherwise incompatible) internal systems. Unlike HTML, XML separates the formatting of a document from its content. Thus, workers can display XML purchase orders in presentations specific to their internal needs or practices.

A fourth approach to dealing with heterogeneous platforms is to add a service provider who either acts as an intermediary or centralizes the application.

Example Some companies have created a common intermediary exchange as a place to procure products and services from a set of suppliers. Sometimes this is done on an industry basis, as in the automotive industry (covisint 2000). Considering the latter case, the exchange does not directly overcome platform and application incompatibilities among the participating organizations, but it does make the challenges considerably more manageable. To see this, suppose there are n distinct firms involved in the exchange. The intermediary has to deal with these n firms separately. Without the intermediary, each firm would have to work out similar

interoperability issues with each of the $n - 1$ other firms, or in total there would be $n \cdot (n - 1)$. such relationships to manage, a much greater total burden. Interestingly, intermediaries tend to compete as well, leading to k intermediaries and thus $k - n$ relationships. For $k = 1$, the single intermediary can gain substantial market control, and the desire of the coordinated firms to retain agility tends to encourage formation of a competing intermediary. As k approaches n, the benefit of having intermediaries disappears. Market forces thus tend to keep the number of competing intermediaries low.

Distinct types of infrastructure can be identified in the layered architecture of figure 7.5. In classifying a particular type of infrastructure, it is important at minimum to ask two basic questions. First, does this infrastructure support a broad range of applications (in the extreme, all applications), or is it specialized to one segment of the application market? Second, is the infrastructure technology-specific, or does it not matter what underlying technologies are used? Neither of these distinctions is black-and-white; there are many nuances. The answers can also change over time because new infrastructure typically must start with only one or a few applications and then grow to more universal acceptance later. Table 7.3 gives examples of infrastructure categorized by application and technology dependence. Each category in the table suggests a distinct business model. As a result, infrastructure

Table 7.3
Examples of infrastructure Defined by Application and Technology Dependence

	Not Application-Dependent	Particular to One Application Market Segment
Not technology-dependent	The Internet TCP transport layer is widely used by applications desiring reliable, ordered delivery of data. The specific networking technologies present are hidden from TCP and layers above by the Internet IP spanning layer.	Message-oriented middleware supports work flow applications, and the Web supports information access and presentation. Each emphasizes distributed applications across heterogeneous platforms.
Particular to one technology platform	Operating systems are specific to a computer platform, although some (like Linux, Solaris, Windows NT, and Windows CE) have been ported to several. Each is designed to support a wide variety of applications on that platform.	Information appliances typically support a single application, and build that application on a single infrastructure technology platform.

suppliers tend to specialize in one (with a couple of notable exceptions): infrastructure suppliers at the lower layers focus on technology-dependent infrastructure, and those at the higher layers on application-dependent infrastructure. In the extreme of application dependence, some infrastructure may support a specific application but offer open APIs in the hope that other applications come later. In the extreme of technology dependence, an infrastructure may be embedded software bundled with hardware and sold as equipment or an appliance. In this case, the software is viewed as a cost of development, with no expectation of selling it independently.

Application- and technology-independent infrastructure clearly has the greatest market potential as measured by adoptions or unit sales. However, this type offers little opportunity for differentiation as to functions or features. To actually achieve universality, it must be highly standardized and hence commoditized. If it is well differentiated from other options, it is likely struggling for acceptance. Thus, this type of infrastructure probably represents the least opportunity for profit but is nevertheless quite important and beneficial. The current trend is therefore to use community-based development methodologies to create and maintain this type of software (as illustrated by the Samba example earlier; also see section 4.2.4), although not exclusively. Its universal appeal and wide use lend themselves to community-based development. This is leading to new types of software licensing approaches that mix the benefits of community-based development such as open source with commercial considerations such as deriving revenue and profit (see chapter 8).

Example Sun Microsystems has been a leading proponent of community-based development of infrastructure, examples being its Java middleware layer for portable execution (see section 4.4.3) and Jini, a platform for plug-and-play interoperability among information appliances (Sun microsystems 1999b). Other firms, such as Apple Computer (Mac OS X) and Netscape (Mozilla browser) have followed this approach. Several firms (e.g., IBM and Hewlett-Packard) have chosen open source Linux as an operating system platform.

At the other extreme, application- and technology-dependent infrastructure is characteristic of the stovepipe architecture (see section 7.1.3), and for the reasons discussed earlier is disappearing because of shrinking market opportunity. Most commercial infrastructure differentiates itself in the application or technology space, but not both.

7.1.6 Competition and Architecture

This section has enumerated some global architectural alternatives and their relation to industry structure. Architecture, because it defines the boundaries of competition and complementarity, is an important strategic issue for software suppliers (see section 6.1), and successfully defining and promulgating an architectural model is an important element of long-term success (Ferguson and Morris 1994). In contrast, suppliers who provide point solutions or who must plug solutions into an architecture defined elsewhere lose some control over their own destiny and have less strategic maneuvering room. Doubtless because of its significance, architectural control has also been a source of industry controversy and even government antitrust complaints (see chapter 8).

Within a given architecture, competition exists at the module level, and where a supply chain is created by hierarchical decomposition, at the submodule level as well. On the other hand, interacting modules are complementary. Suppliers naturally try to avoid direct competition in modules, particularly because high creation costs and low replication and distribution costs for software make competitive pricing of substitutes problematic (see chapter 9). Generally, open standardized interfaces (see section 7.2.3) make head-on competition more difficult to avoid, and for this reason industry standards are increasingly defined with a goal of enabling competitive suppliers to differentiate themselves with custom features or extensions while maintaining interoperability. In contrast, if suppliers choose not to offer complementary modules themselves, they encourage competitive options for those modules so that customers have a wider range of options with attractive prices and features and so that the overall system pricing is more attractive.

The architectural options and evolution discussed here have considerable implications for competition. Applications and infrastructure are complementary, and thus suppliers of each generally encourage competitive options in the other. While the expansion of infrastructure capabilities is a way to improve economic efficiency through sharing (see chapter 9) and to improve performance and quality, it also changes the competitive landscape by shrinking some application markets or at least introducing new competition.

No architectural issue has had more effect than the evolution from stovepipe toward layering architecture. This shifts the landscape from horizontal market segmentation, with an inclination toward vertical integration within each segment, to a vertical segmentation of functions where multiple complementary suppliers must cooperate to realize a full system solution.

The spanning layer is a key element of a layered architecture. Because it allows greater independence of evolution in the layers below and above, it is another form of common intermediary that makes practical a larger diversity of interoperable options below and above. Even if not initially defined with open interfaces, the ubiquity of a spanning layer implies that its interfaces below and above become de facto standards. In order for the entire infrastructure to evolve, a spanning layer and its interfaces must also evolve over time, insofar as possible through extensions rather than changes. While the spanning layer offers compelling advantages, commercial control of such a layer raises complaints of undue influence over other layers and overall system evolution. One response is government-imposed limits on the business practices of the owner of a spanning layer (see chapter 8). Another is to abandon the spanning layer in a way that preserves most advantages, such as horizontal heterogeneity within a layer, while maintaining interoperability or portability (see sections 7.1.5 and 7.3). Another is to promulgate a spanning layer in the public domain (through community-based development or similar methodologies, as in the Samba example).

7.2 Cooperation in the Software Industry

Just as participants in the software value chain (including nonsoftware companies) maintain ongoing business relationships (see section 6.3), so do participants within the software creation industry. Monolithic software solutions are today the exception rather than the rule; in most cases, a total solution of value to users integrates content from a number of software companies.

Example A single desktop computer with a standard suite of office software might serve the needs of some users, and the software on such a platform might come from a single supplier like Apple Computer, Microsoft, or Sun Microsystems. Even in this simple case there will likely be contributions from other suppliers. For example, Microsoft Word XP includes modules and content acquired from other suppliers, like the equation editor, the document version comparer, parts of the spelling correction system, thesaurus, hyphenators, and dictionaries for many different languages, as well as some templates and fonts. Further, when the user browses the Web, the Web server may well be open source software (like Apache) or proprietary Web server software from another supplier (like IBM WebSphere or BEA WebLogic).

A pervasive issue is the coordination of software suppliers so that their solutions are either automatically composable or can at least be purposefully integrated. This

creates a conspicuous need and opportunity for different software companies to coordinate through business or cooperative arrangements. As in other industries, coordination can take many forms, the extremes being proprietary bilateral business relationships on the one hand, and cooperative standards processes open to all interested parties on the other.

7.2.1 Supplier-Consumer Relationships

Some business relationships within the software industry take the traditional supplier-consumer form, although this does not evoke standard images of warehouses, shipping, and inventory. Since software can be freely replicated, a supplier need only provide a single copy to the customer together with the appropriate authorization, in the form of a licensing agreement (see chapter 8) spelling out the terms and conditions, for the customer to replicate the software in its products or to provision within its environment.

Where one software supplier is incorporating modules supplied by others, those modules must be integrated. This system integration step frequently requires modification of the purchased modules. The license permitting, changes to acquired modules may be made by the integrator (this requires source code). More often, the supplier makes these changes, and these repairs or refinements benefit all customers. Generally, the process and issues to be addressed are similar to end-user acquisition of software (see section 6.3.4). Differences do arise if the customer passes this software through to its own customers rather than provisioning it internally. Thus, the revenue stream expected from its customers, rather than its internal value proposition, becomes an element of price negotiation. Further, there is the issue of customer support: How do operators and users obtain support for modules acquired rather than developed by their immediate supplier? Is this a two-step process, or should the supplier directly support operators and users? Of course, customers generally prefer an integrated customer support solution. A common form of licensing agreement refers to the indirect suppliers as original equipment manufacturers (OEMs) and leaves all customer support with the immediate supplier of a product. Internally, that supplier will draw on technical support from the OEM.

Example Comparable OEM agreements exist between manufacturers of personal computers and the suppliers of preinstalled operating systems and applications. Customers receive integrated support from the computer manufacturer, who may in turn depend on the OEM when it cannot deal with an issue.

7.2.2 Application Program Interface

Recall that the API is an interface designed to support a broad class of extensions (see section 4.3.4). The open API allows one software supplier to extend or incorporate software from another supplier without establishing a formal business relationship. The owner of the open API exports services through an interface that is documented and where the software license allows for the unfettered use of this interface unconstrained by intellectual property restrictions and without payment. Technically, it is possible for a module from another supplier to invoke actions at this interface, which requires that it be accessible through mechanisms embodied in industry-standard infrastructure. One of the roles of infrastructure is to enable the composability of modules from different suppliers, and the API is one of the key constructs made possible.

Example An application may offer an API that allows other suppliers to add value to that application, for instance, in the common gateway interface to a Web server that allows other applications to display their content via a Web browser. This API is technically feasible because the operating system provides mechanisms for one program to interact with another program executing on the same host.

It should be emphasized that not all interfaces in a software product are APIs. Most interfaces are proprietary, designed for internal interaction of modules and neither documented nor made available through technical means to software from other suppliers. Other interfaces may be documented and technically available, but because they are designed for a specific and narrow purpose, they fail to quality as an API. Further, suppliers reserve the right to change internal interfaces but implicitly or explicitly commit to extending but not changing an API (so as not to break other modules depending on it). Choosing to include an API in a software product is a serious business decision. Besides potential benefits, there are significant costs. Future versions of the product will either have to maintain compatibility, thus possibly requiring substantial engineering effort, or abandon existing clients using the API, thus risking dissatisfied customers and opening an opportunity for competitors.

If future extensions can be anticipated and are of broad interest, the supplier may wish to create and market these extensions itself, rather than ceding this opportunity to other suppliers, by building in an API. Infrastructure software's value is through the applications supported, and the API is the enabler. To the application software supplier, the API may be a vehicle by which other suppliers or even customers may customize that application to more specific (company or vertical industry) needs, increasing its value.

An alternative to the API is to offer contract development services to customize software. The supplier may maintain a services organization that contracts for customizations or extensions to meet specific customer needs.

The API is a software interface for software executing within a single host. A similar idea can be achieved over the network, where software from one supplier can interface with software from another supplier using the network. In this case, the API is replaced by a network protocol with similar business issues and characteristics.

7.2.3 Open Standards

An *industry standard* is a specification that is commonly agreed upon, precisely and completely defined, and well documented so that any supplier is free to implement and use it. Of course, it may or may not be widely adopted or uniformly implemented. In the software industry, the most common targets for standardization are architectural decomposition and the interfaces or network protocols defining the interactions of modules within that architecture. This type of standard seeks interoperability among modules, either within the same host or across the network (see section 4.5).

Example The USB (universal serial bus) port is a standard interface to personal computer peripherals that includes physical (plug geometry, functions of the different pins) and electrical (voltage and waveform) characteristics, as well as the formats of bit streams that allow messages to be passed between a CPU and a peripheral. Like most standards, it does not address the complementarity of function in the computer and the peripheral, such as printing a page or communicating over a telephone line.

Another common target for standardization is the representation used for specific types of information, so that information can be exchanged among different applications or within an application.

Example JPEG and MPEG are popular industry-standard compressed representations for pictures and audio/video, respectively. They allow one application to capture music in a file and another application to access that file and recreate the music. MP3 is a popular standard for sharing compressed music based on the audio portion of MPEG.

Programming languages are often standardized as well.

Example The International Standards Organization (ISO) has standardized COBOL, FORTRAN, PL/1; the American National Standards Institute (ANSI) has

standardized C and C^{++}; the European Computer Manufacturers Association (ECMA) has standardized ECMAscript (also known as JavaScript or JScript) and $C^{\#}$.

An *open standard* is available for anybody to implement, well documented, and unencumbered by intellectual property restrictions, so any supplier is free to implement the standard without making prior business arrangements.

Example Many open standards are created by independent standardization bodies in which a number of companies collaborate in finding a mutually satisfactory solution. The body that creates the open Internet standards (including IP, UDP, and TCP) is the Internet Engineering Task Force (IETF). The World Wide Web Consortium (W3C) defines open standards for the evolution of the Web.

"Openness" is not an absolute because some of these properties can be relaxed, making the standard less open but still not closed or proprietary.

Example Sometimes a standard encumbered by intellectual property rights may be considered open if a promise has been made to exercise those rights in a measured fashion. In the most common arrangement, in return for inclusion in the standard the owner promises that a license will be granted to any and all under terms that are reasonable (moderate in cost) and nondiscriminatory (the same terms for all). For instance, the MP3 audio compression standard is an ISO/IEC standard, but is still covered by patents held by Fraunhofer IIS-A and Thomson multimedia that require licensing and fee payment for any but private and small-scale use.[7]

Other interfaces, protocols, or representations may carry more restrictions and still be labeled an industry standard, even if not considered an open standard.

Example Java is promulgated as a programming language, associated virtual machine for supporting portable execution, and an environment for portable and mobile code (see section 4.4). The specifications and associated tools were first developed by Sun Microsystems, which maintained licensing terms intended to prevent variants. Sun imbued Java with the characteristics of a standard (widely promulgated and used) while retaining control through intellectual property laws (see chapter 8). Among those provisions, any implementations that use the Java trademark must meet certain acceptance tests.

7.2.4 Why Standards?

The industry standard helps coordinate suppliers of complementary products, but it is not the only such mechanism. The supplier-customer business relationship allows a supplier to buy rather than make some portion of its software product.

The API enables a one-to-many relationship, where one software supplier deliberately creates an opportunity for all other suppliers to extend or exploit its product without the need for a formal business relationship. The industry standard excels at supporting a multilateral relationship among suppliers. The typical approach is to define and document an interface or a network protocol that can be exploited by many companies. In contrast to the API, where one supplier maintains a proprietary implementation of one side of an interface and allows other suppliers to define products on the other side of that interface, a standardized interface allows companies to define products that support the interface from either side.

From the customer and societal perspectives, open standards allow competition at the subsystem level: suppliers can create competitive substitutes for subsystems and know that the customer will have available the necessary complementary subsystems from other suppliers to forge a complete solution. Similarly, customers can mix and match subsystems from different suppliers if they feel this results in a better overall solution in dimensions such as price, features, performance, and quality. Modules can be replaced without replacing the entire system, reducing switching costs and lock-in. The disadvantage is that the customer must integrate subsystems from different vendors. In spite of standards, this additional integration takes time and effort, and sometimes introduces problems.

Example The PC offers open standards for serial and parallel connections between CPU and peripherals, including modems, printers, and display, so customers can mix and match PCs and peripherals from different manufacturers. Apple Computer pursued a more monolithic approach with the original Macintosh, which had the advantage that the system was operational out of the box. As the PC platform has matured, plug-and-play technology has made integration more seamless, and vendors like Dell accept customized system orders and perform the integration for the customer. Meanwhile, the Macintosh has moved toward an open standards approach (supporting open industry standards such as the universal serial bus). Today this distinction between the two platforms is minimal.

Network effects sometimes drive standardization (see section 3.2.3) in a multivendor solution. The incentive for standardization in this case is to avoid the proliferation of multiple networks, with the resulting fragmentation and reduced value to users and the benefits of positive feedback.

Example The peer-to-peer architecture for distributed applications creates a need for standards to support interoperability among peers (see section 4.5.3). Without such a standard, users could only participate in the application with users who have

adopted that same vendor's solution. This is illustrated by instant messaging, where several firms offer services (AOL, Microsoft, and Yahoo, among others) that are currently incompatible, creating fragmented networks. The DVD illustrates the benefit of a standardized information representation that tames indirect network effects. Two industrial consortiums proposed incompatible standards for video playback but ultimately negotiated a single standard, driven by concern about the market dampening effect of network effects and consumer confusion if two or more standards were marketed, and by pressure from content suppliers, who were concerned about these issues.

7.2.5 How Standards Arise

An industry standard is the outcome of a process, sometimes a long and messy one. Influences on the eventual standard may be user needs, market forces, the interests of or strategy pursued by individual suppliers, and occasionally government laws or regulations. The possibilities range from a *de facto standard* to a *de jure standard*. The de facto standard begins life as a proprietary interface or protocol, but through market forces becomes so commonly adopted by many companies that it is an industry standard in fact (Shapiro and Varian 1999a). In the case of interfaces, some de facto standards begin life as APIs. Undocumented proprietary interfaces are less likely to become de facto standards because they prohibit (using intellectual property laws) or discourage participation by other suppliers.

Example The Hayes command set started as an API chosen by a leading voiceband modem manufacturer and was initially offered by most suppliers of telecommunications software to control the Hayes modem. Since this API was widely supported by software, other modem manufacturers began to implement the same API, and it became a de facto standard. Later, Hayes attempted to force other modem manufacturers to pay royalties based on patented technology it had incorporated into the implementation. Another example is the operating system APIs, allowing programs to send a packet over the network or save a file to disk. The primary purpose is encouraging application software suppliers to build on the operating system; a diversity of applications provides greater value to users. A side effect is to potentially enable an alternative infrastructure software supplier to independently implement and sell a direct substitute operating system, except to the extent that the API may be protected by intellectual property restrictions (see chapter 8.) Such independent reimplementation is unlikely for an operating system, however, because of the large investment and unappealing prospect of head-to-head

competition with an entrenched supplier with substantial economies of scale (see chapter 9).

In the case of both interfaces and protocols, de facto standards often begin as an experimental prototype from the research community.

Example The protocol and data format used to interact between client and server in the Web (HTTP and HTML) allows a Web browser and server to compose regardless of who supplies the client and server. It began as a way to share documents within a research community. Later, it was popularized by the innovative Mosaic Web browser from the University of Illinois, which provided a general graphical user interface. Today, there are more than a half-dozen suppliers of servers and browsers, and within the limits of the imprecise definitions of HTML, any server can interoperate with any browser using these standards (and their successors). Similarly, the socket is an operating system API that allows applications to communicate over the network. It has become a de facto standard resident in several operating systems, but it started as an API for the Berkeley UNIX operating system from the University of California.

At the other end of the spectrum, the de jure standard is sanctioned by a legal or regulatory entity.

Example Regulatory forces are most likely to impose themselves when some public resource like the radio spectrum is required. In most countries there is a single legally sanctioned standard for radio and television broadcasting, as for wireless telephony. In the latter case the United States is an exception; the Federal Communications Commission specifically encouraged the promulgation of several standards. These standards deal with the representation and transmission of voice only across the wireless access link, and admit the conversions that allow for end-to-end voice conversations; direct network effects do not intervene. Another example is the Ada programming language, defined by the U.S. Department of Defense and imposed on its contractors until the late 1990s.

There are many cases intermediate to de facto and de jure, some of which are discussed later in conjunction with standards processes.

As applied to interfaces and network protocols, an essential first step in defining such standards is to locate interfaces or protocols that are candidates for standardization. This is an architectural design issue for purposes of standardization as well as implementation. There are several approaches to determining where there should be an interface or protocol to standardize. The first is to explicitly define the location of an interface as part of the standardization process. Such decomposition is

called a *reference model*, a partial software architecture covering aspects of the architecture relevant to the standard. A reference model need not be a complete architecture; for example, modules within an implementation may be hierarchically decomposed from a single reference-model module, an implementation choice not directly affecting compliance with the standard.

Example CORBA is a standard for a middleware infrastructure supporting object-oriented distributed systems promulgated by the Object Management Group. One of its primary contributions is a reference model for a number of common services that support such distributed applications.

A second approach to defining the location of a standardized interface is creating an interface and putting it into the public domain as a standard or letting it grow into a de facto standard.

Example Desktop computer vendors (both IBM and Apple) defined a number of interfaces that grew into industry standards, including interfaces to monitor and peripherals, an API for application programs, and standards for the bus that supports expansion cards.

Third, the location of an open interface might be defined by market dynamics or be a side effect of the preexisting industrial organization. These types of interfaces typically follow the lines of core competencies, such as integrated circuit manufacture and infrastructure of application software.

Example When IBM designed its first PC, it made decisions on outside suppliers that predefined some interfaces within the design, and those interfaces later evolved into de facto standards. By choosing an Intel microprocessor rather than developing its own, IBM implicitly chose an instruction set for program execution. By deciding to license its operating system (MS-DOS) from Microsoft (which importantly targeted this instruction set) rather than develop its own, IBM adopted operating system APIs that were later used by alternative operating systems (for example, Linux uses a FAT32 file system adopted from DOS, and Novell marketed a version of DOS). These choices reduced the time to market but also created an opportunity for other suppliers, including the PC clone manufacturers (Compaq was an early example) and AMD (which manufactures microprocessor chips compatible with Intel's).

Standards also address a serious problem in software engineering. In principle, a new interface could be designed whenever any two modules need to compose. However, the number of different interfaces must be limited to contain the development and maintenance costs arising from a proliferation of interfaces. Besides

this combinatorial problem, there is the open-world problem. The open-world assumption in systems allows new modules to be added that weren't known or in existence when the base system was created—this is the motivation for APIs. It is impractical (indeed impossible) to have a complete set of special-case or proprietary interfaces to connect a full range of modules that may arise over time. A practical alternative is to define a limited set of standardized interfaces permitting interoperability over a wide range of functionality and complementarity.

Example The CORBA standards standardize IIOP, a network protocol layered on top of TCP, which allows modules (in this case, the most limited case of objects) to interface with one another in a similar way whether they reside on the same host or different hosts. In effect, IIOP hides the details of the underlying network protocols (potentially multiple) and multiple platform implementations of those protocols behind a familiar interface. While individual applications would be free to develop a similar capability on a proprietary basis, an industry-standard solution reduces the number of implementations that are developed and maintained.

7.2.6 The Evolution of Standards Processes

Interfaces, the functionality related to these interfaces, the preferred decomposition of systems, and the representations used for sharing information can all be standardized to enable interoperability. For needs that are well understood and can be anticipated by standardization bodies (such as industrial consortiums or governmental standardization institutions) standards can be forged in advance of needs and later implemented by multiple vendors. This process has unfortunately not worked well in the software industry because of the rapid advances made possible by software's inherent flexibility and rapid distribution mechanisms, with the result that new products are often exploring new technology and applications territory. Thus, this industry has relied heavily on de facto standardization.

Another approach has been to emphasize future extensibility in standards that are developed. This is a natural inclination for software, which emphasizes elaboration and specialization of what already exists (e.g., through layering; see section 7.1.3). For example, it is often feasible to elaborate an existing API rather than to define a new one. This can be accomplished by following the *open-closed principle*, which requires that interfaces be open to extension but closed to change. As long as existing actions are unchanged, the interface can be elaborated by adding new actions without affecting modules previously using the interface.

Example Successive generations of an operating system try to maintain compatibility with existing applications by not changing the actions available in its API.

The new version may be new or improved "under the hood," for example, improving its stability or performance without changing the API. The new version may add new capabilities (added actions) to benefit future applications without changing those aspects of the API used by old applications.

The Internet has increased the importance of standards because of the new dependence (and direct network effects) it creates across different platforms. In an attempt to satisfy this thirst for standards, but without introducing untoward delay and friction in the market, industry has experimented with more facile and rapid standardization processes.

One trend is standardization processes well integrated with a research or experimental endeavor, in which the standard is deliberately allowed to evolve and expand in scope over time based on continual feedback from research outcomes and real-world experience. In fact, this type of standardization activity shares important characteristics (like flexibility and user involvement) with agile software development processes (see section 4.2.5).

Example IETF has always recognized that its standards are a work in progress. The mechanism is to publish and than never change specific standards but to allow newer versions to make older ones obsolete. Most IETF standards arise directly from a research activity, and there is a requirement that any additions to the suite of standards be based on working experimental code. One approach used by the IETF and others is to rely initially on a single implementation that offers open-world extension hooks. Once it is better understood, a standard may be lifted off the initial implementation, enabling a wider variety of interoperable implementations.

In contrast to this continual refinement, a traditional top-down process is less chaotic and allows greater reflection on the overall structure and goals. It attempts to provide a lasting solution to the whole problem, all at once. A refinement process acknowledges that technologies are dynamic; whereas a reference architecture must be reasonably well defined to begin with, the details of functionality and interfaces can evolve over time.

Much depends on the maturity of an industry. For the time being, the de facto and continual refinement standardization processes are appropriate for many aspects of software because they allow innovation and evolution of solutions, reflecting market realities. When a stage of maturity is reached where functionality is better defined and stable, traditional top-down standardization processes can take over.

Layering is important because it allows standards to be built up incrementally rather than defined all at once (see section 7.1.3). The bottom layer (called wiring or plumbing standards) is concerned with simple connection-level standards.

Functionality can then be extended one layer at a time, establishing ever more elaborate rules of interoperation and composability.

Example The early Internet research, as more recently the IETF, used layering. The bottom layer consisted of existing local-area networking technologies and displayed horizontal heterogeneity because there were numerous local-area and access networking technologies. The Internet standard added an IP layer interconnecting these existing technologies, and it provides today a spanning layer supporting a number of layering alternatives above. The IETF has systematically added layers above for various specific purposes. Sometimes lower layers need to be modified. For example, version four of the IP layer is widely deployed today, and the next version (version six) has been standardized. Because IP is widely used, any new version should satisfy two key constraints if at all possible. First, it should coexist with the older version, since it is impractical to upgrade the entire network at once. Second, it should support existing layer implementations above while offering new services or capabilities to new implementations of those layers or to newly defined layers.

Another trend is standardization processes that mimic the benefits of de facto standards but reduce or eliminate the time required for the marketplace to sort out a preferred solution. A popular approach is for a group of companies to form a consortium (often called a forum) that tries to arrive at good technical solutions by pooling expertise; the resulting solutions do not have the weight of a formal standard but rather serve as recommendations to the industry. Often such a consortium will request proposals from participants and then choose a preferred solution or combine the best features of different submissions or ask that contributors work to combine their submissions. Member companies follow these standards voluntarily, but the existence of these recommendations allow the market to arrive at a de facto standard more quickly.

Example The Object Management Group, the developer of the CORBA standards, was formed to develop voluntary standards or best practices for infrastructure software supporting distributed object-oriented programs. It now has about 800 member organizations. W3C was formed by member companies at the Massachusetts Institute of Technology to develop voluntary standards for the Web; it now has more than 500 member organizations. ECMA was formed to reach de facto standards among European companies but has evolved into a standards body that offers a fast track to the International Organization for Standardization.

7.2.7 Minimizing the Role of Standards

While standardization has many benefits, they have disadvantages as well. In an industry that is changing rapidly with robust innovation, the existence of standards and the standardization process can impede technical progress. Sometimes standards come along too late to be useful.

Example The Open Systems Interconnect (OSI) model was a layered network protocol providing similar capabilities to the Internet technologies. It was an outgrowth of a slow international standardization process and, because it attempted to develop a complete standard all at once, was expensive and slow to be implemented as well. By the time it arrived, the Internet had been widely deployed and was benefiting from positive feedback from network effects. OSI was never able to gain traction in the market.

Where standards are widely adopted, they can become a barrier to progress. This is an example of lock-in of the entire industry resulting from the difficulty and expense of widely deploying a new solution.

Example Version six of IP has been much slower to deploy than expected. While it will likely gain acceptance eventually, version four has been incrementally upgraded to provide many of the capabilities emphasized in version six, and the substantial trouble and expense of deploying version six is an obstacle.

Another disadvantage of standards is that they may inhibit suppliers from differentiating themselves in the market. A way to mitigate this, as well as to allow greater innovation and faster evolution of the technology, is to define flexible or extensible standards.

Example XML is a W3C standard for representing documents. Originally defined as a replacement for HTML in the Web, XML is gaining momentum as a basis for exchanging information of various types among departmental, enterprise, and commerce applications, and is one underpinning of Web services (see section 7.3.7). One advantage is that unlike HTML, it separates the document meaning from screen formatting, making it useful to exchange meaningful business documents whose content can be automatically extracted and displayed according to local formatting conventions. Another advantage is its extensibility, allowing new industry- or context-specific representations to be defined. XML and its associated tools support a variety of context-specific standards or proprietary representations.

Where reasonable to do so, it is appropriate to minimize or eliminate the role of standards altogether. Although standards are always necessary at some level,

modern software technologies and programmability offer opportunities to reduce their role, especially within applications (as opposed to infrastructure).

Example The device driver shown in figure 7.8 is used in connecting a peripheral (like a printer) to a personal computer. The idea is to exploit the programmability of the computer to install a program that communicates with the printer, with complementary embedded software in the printer. This allows the operating system to focus on defining standard high-level representations for printed documents (such as Postscript), while the device driver encapsulates low-level protocols for interoperation between the computer and the printer. Since the device driver is supplied by the printer manufacturer, it can do whatever it chooses (like differentiating one printer from another) without requiring an interoperability standard. Of course, the device driver and printer build on a standard for exchanging content-blind messages, such as the computer serial or parallel port.

Mobile code can realize the same idea dynamically (see figure 7.9). Interoperability issues suggest the need for standardization when two modules on different

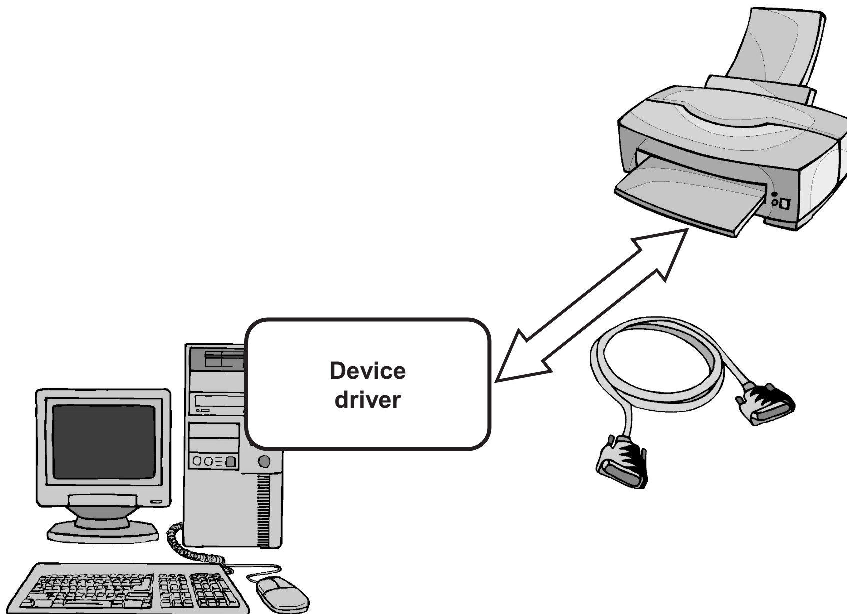

Figure 7.8
The device driver can allow interoperability while moving standardizations to a higher abstraction.

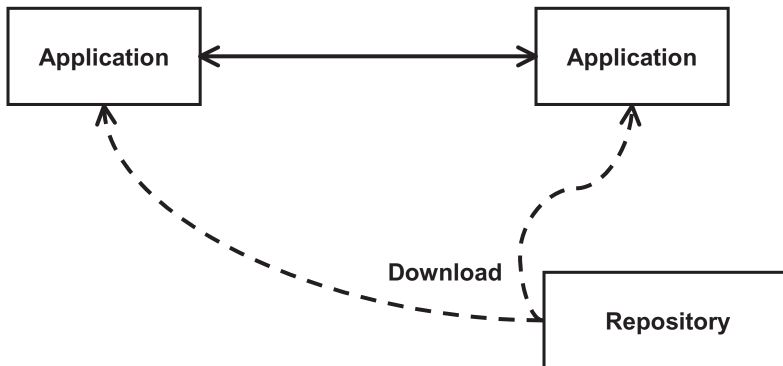

Figure 7.9
Direct network effects can be eliminated by mobile code.

hosts may originate with different suppliers. However, if both modules originate with the same supplier, they may be interoperable by construction with no need for standardization. Their interfaces can even be changed in new versions, as long as both modules are upgraded simultaneously.

Example Real Networks supplies complementary streaming audio-video RealServer and a RealPlayer for the client desktop. Over time, Real has been relatively free to upgrade its RealServer capabilities, even at the expense of compatibility with the RealPlayer, because it is reasonable to expect users to upgrade the client to the latest available version over the network. (Users with no Internet connection would not be candidates to use streaming audio-video.)

Downloaded software or mobile code is a particularly powerful way to bypass direct network effects, as evident in a peer-to-peer architecture.

Example Successful peer-to-peer applications like Napster (music file sharing) and Groove (file sharing and collaborative tools) have benefited from the ability to download the peer software from a central server. To join the network, a new user can easily download and install the necessary software (or with mobile code it can even be made transparent). Were it necessary to purchase equipment or software in a store, these sorts of applications would find it dramatically more difficult to reach critical mass and benefit from positive feedback.

Another approach to mitigating some problems with standardization is to standardize languages that can describe application elements, such as the interaction between modules, the functionality of modules, or the representation of information elements. We call this a meta-standard because it standardizes a way of

describing something rather than standardizing that something directly. This can substantially increase the ability of suppliers to innovate and differentiate.

Example For communicating images from one host to another, a representation that includes a way of digitizing the image and compressing it must be shared by the transmitter and receiver. To avoid standardization, a meta-standard might take the form of a language capable of describing a large collection of decompression algorithms. A typical description of a compression algorithm would be something like "use an *n* by *n* discrete cosine transform with $n = 8$ followed by a quantization algorithm of the following form. . . ." Constrained only by the linguistic expressiveness of the meta-standard language, a transmitter is free to choose any compression algorithm and convey it (this is a form of mobile code) along with the image representation to the receiver. An early example is self-extracting archives, which are compressed collections of computer files arriving as an executable bundle that, upon execution, unpacks itself, yielding the collection of files.

Rudimentary forms of meta-standards already exist.

Example The interface definition language (IDL) of CORBA allows modules to disclose their capabilities by describing the actions that are available. XML for documents provides a language for describing, in effect, new markup languages.

7.3 Component Software

Most new software is constructed on a base of existing software (see section 4.2). There are a number of possibilities as to where the existing software comes from, whether it is modified or used without modification, and how it is incorporated into the new software. Just as end-users face the choice of making, buying, licensing, or subscribing to software, so do software suppliers. This is an area where future industry may look very different (Szyperski 1998).

7.3.1 Make vs. License Decisions

Understanding the possible sources of an existing base of software leads to insights into the inner workings of the software creation industry. Although the possibilities form a continuum, the three distinct points listed in table 7.4 illustrate the range of possibilities. In this table, a software development project supports the entire life cycle of one coherent system or program through multiple versions (see section 5.1.2).

The handcrafting methodology was described in section 4.2. In its purest form, all the source code for the initial version is created from scratch for the specific

Table 7.4
Methodologies for Building Software on an Existing Code Base

Development Methodology	Description
Handcrafting	Source code from an earlier stage of the project is modified and upgraded to repair defects and add new features.
Software reuse	In the course of one project, modules are developed while anticipating their reuse in future projects. In those future projects, source code of existing modules from different projects is modified and integrated.
Component assembly	A system is assembled by configuring and integrating preexisting components. These components are modules (typically available only in object code) that cannot be modified except in accordance with built-in configuration options, often purchased from another firm.

needs of the project and is later updated through multiple maintenance and version cycles. Thus, the requirements for all the modules in such a handcrafted code base are derived by decomposition of a specific set of system requirements.

In the intermediate case, software reuse (Frakes and Gandel 1990; Gaffney and Durek 1989), source code is consciously shared among different projects with the goal of increasing both organizational and project productivity (recall the distinction made in section 4.2.3). Thus, both the architecture and requirements for at least some individual modules on one project anticipate the reuse of those modules in other projects. A typical example is a *product line architecture* (Bosch 2000; Jacobson, Griss, and Jönsson 1997), where a common architecture with some reusable modules or components is explicitly shared (variants of the same product are an obvious case; see section 5.1.2). To make it more likely that the code will be suitable in other projects, source code is made available to the other projects, and modifications to that source code to fit distinctive needs of the other project are normally allowed. Because it is uncommon to share source code outside an organization, reuse normally occurs within one development organization. A notable exception is contract custom development, where one firm contracts to another firm the development of modules for specific equirements and maintains ownership in and source code from that development outcome.

Example A hypothetical example illustrates software reuse. The development organization of Friendly Bank may need a module that supports the acquisition, maintenance, and accessing of information relative to one customer. This need is

first encountered during a project that is developing and maintaining Friendly's checking account application. However, Friendly anticipates future projects to develop applications to manage a money market account and later a brokerage account. The requirements of the three applications have much in common with respect to customer information, so the first project specifically designs the customer module, trying to meet the needs of all three applications. Later, the source code is reused in the second and third projects. In the course of those projects, new requirements are identified that make it necessary to modify the code to meet specific needs, so there are now two or three variants of this customer module to maintain.

Reused modules can be used in multiple contexts, even simultaneously. This is very different from the material world, where reuse carries connotations of recycling, and simultaneous uses of the same entity are generally impossible. The difference between handcrafted and reusable software is mostly one of likelihood or adequateness. If a particular module has been developed with a special purpose in mind, and that purpose is highly specialized or the module is of substantial but context-specific complexity, then it is unlikely to be reusable in another context. Providing a broad set of configuration options that anticipates other contexts is a way to encourage reuse.

The third option in table 7.4 is component assembly (Hopkins 2000; Pour 1998; Szyperski 1998). In this extreme, during the course of a project there is no need to implement modules. Rather, the system is developed by taking existing modules (called components), configuring them, and integrating them. These components cannot be modified but are used as is; typically, they are acquired from a supplier rather than from another project within the same organization. To enhance its applicability to multiple projects without modification, each component will typically have many built-in configuration options.

Example The needs of Friendly Bank for a customer information module are hardly unique to Friendly. Thus, a supplier software firm, Banking Components Inc., has identified the opportunity to develop a customer information component that is designed to be general and configurable; in fact, it hopes this component can meet the needs of any bank. It licenses this component (in object code format) to any bank wishing to avoid developing this function itself. A licensing bank assembles this component into any future account applications it may create.

Although the software community has seen many technologies, methodologies, and processes aimed at increasing productivity and software quality, the consensus today is that component software is the most promising approach. It creates a supply

chain for software, in which one supplier assembles components acquired from other suppliers into its software products. Competition is shifted from the system level to the component level, resulting in improved quality and cost options.

It would be rare to find any of these three options used exclusively; most often, they are combined. One organization may find that available components can partly meet the needs of a particular project, so it supplements them with handcrafted modules and modules reused from other projects.

7.3.2 What Is a Component?

Roughly speaking, a *software component* is a reusable module suitable for composition into multiple applications. The difference between software reuse and component assembly is subtle but important. There is no universal agreement in the industry or literature as to precisely what the term *component* means (Brown et al. 1998; Heinemann and Councill 2001). However, the benefits of components are potentially so great that it is worthwhile to strictly distinguish components from other modules and to base the definition on the needs of stakeholders (provisioners, operators, and users) and the workings of the marketplace rather than on the characteristics of the current technology (Szyperski 1998). This leads us to the properties listed in table 7.5. Although some may still call a module that fails to satisfy one or more of these properties a component, important benefits would be lost.

One of the important implications of the properties listed in table 7.5 is that components are created and licensed to be used as is. All five properties contribute to this, and indeed encapsulation enforces it. Unlike a monolithic application (which is also purchased and used as is), a component is not intended to be useful in isolation; rather its utility depends on its composition with other components (or possibly other modules that are not components). A component supplier has an incentive to reduce context dependence in order to increase the size of the market, balancing that property against the need for the component to add value to the specific context. An additional implication is that in a system using components (unlike a noncomponentized system) it should be possible during provisioning to mix and match components from different vendors so as to move competition from the system level down to the subsystem (component) level. It should also be possible to replace or upgrade a single component independently of the remainder of a system, even during the operation phase, thus reducing lock-in (see chapter 9) and giving greater flexibility to evolve the system to match changing or expanding requirements.[8] In theory, the system can be gracefully evolved after deployment by incrementally upgrading, replacing, and adding components.

Table 7.5
Properties That Distinguish a Component

Property	Description	Rationale
Multiple-use	Able to be used in multiple projects.	Share development costs over multiple uses.
Non-context-specific	Designed independently of any specific project and system context.	By removing dependence on system context, more likely to be general and broadly usable.
Composable	Able to be composed with other components.	High development productivity achieved through assembly of components.
Encapsulated	Only the interfaces are visible and the implementation cannot be modified.	Avoids multiple variations; all uses of a component benefit from a common maintenance and upgrade effort.
Unit of independent deployment and versioning	Can be deployed and installed as an independent atomic unit and later upgraded independently of the remainder of the system[a]	Allows customers to perform assembly and to mix and match components even during the operational phase, thus moving competition from the system to the component level.

a. Traditionally, deployment and installation have been pretty much the same thing. However, Sun Microsystem's EJB (a component platform for enterprise applications) began distinguishing deployment and installation, and other component technologies are following. Deployment consists of readying a software subsystem for a particular infrastructure or platform, whereas installation readies a subsystem for a specific set of machines. See Szyperski (2002a) for more discussion of this subtle but important distinction.

While a given software development organization can and should develop and use its own components, a rather simplistic but conceptually useful way to distinguish the three options in table 7.4 is by industrial context (see table 7.6). This table distinguishes modules used within a single project (handcrafted), within multiple projects in the same organization (reusable), and within multiple organizations (component).

Component assembly should be thought of as *hierarchical composition* (much like hierarchical decomposition except moving bottom-up rather than top-down).[9] Even though a component as deployed is atomic (encapsulated and displaying no visible internal structure), it may itself have been assembled from components by its supplier, those components having been available internally or purchased from other suppliers.[10] During provisioning, a component may be purchased as is, configured for the specific platform and environment (see section 4.4.2), and assembled and integrated with other components. As part of the systems management func-

Table 7.6
Industrial Contexts for Development Methodologies

Methodology Type	Industrial Context	Exceptions
Handcrafted	Programmed and maintained in the context of a single project.	Development and maintenance may be outsourced to a contract development firm.
Reusable	Programmed anticipating the needs of multiple projects within a single organization; typically several versions are maintained within each project.	A common module may be reused within different contexts of a single project. Development and maintenance of reusable modules may be outsourced.
Component	Purchased and used as is from an outside software supplier.	Components may be developed within an organization and used in multiple projects.

tion during operation, the component may be upgraded or replaced, or new components may be added and assembled with existing components to evolve the system.

A component methodology requires considerably more discipline than reuse. In fact, it is currently fair to say that not all the properties listed in table 7.5 have been achieved in practice, at least on the widespread and reproducible basis. Components are certainly more costly to develop and maintain than handcrafted or reusable modules. A common rule of thumb states that reusable software requires roughly several times as much effort as similar handcrafted software, and components much more. As a corollary, a reusable module needs to be used in a few separate projects to break even, components even more.

If their use is well executed, the compensatory benefits of components can be substantial. From an economic perspective, the primary benefit is the discipline of maintenance and upgrade of a single component implementation even as it is used in many projects and organizations. Upgrades of the component to match the expanding needs of one uses can benefit other users as well.[11] Multiple use, with the implicit experience and testing that result, and concentrated maintenance can minimize defects and improve quality. Components also offer a promising route to more flexible deployed systems that can evolve to match changing or expanding needs.

Economic incentives strongly suggest that purely in economic terms (neglecting technical and organizational considerations) components are more promising than reuse as a way to increase software development productivity, and that components

will more likely be purchased from the outside than developed inside an organization. Project managers operate under strict budget and schedule constraints, and developing either reusable or multiuse modules is likely to compromise those constraints. Compensatory incentives are very difficult to create within a given development organization. While organizations have tried various approaches to requiring or encouraging managers to consider reuse or multiple uses, their effectiveness is the exception rather than the rule.

On the other hand, components are quite consistent with organizational separation. A separate economic entity looks to maximize its revenue and profits and to maximize the market potential of software products it develops. It thus has an economic incentive to seek the maximum number of uses, and the extra development cost is expected to be amortized over increased sales. Where reuse allows the forking of different variations on a reused module to meet the specific needs of future projects, many of the economies of scale and quality advantages inherent in components are lost. It is hardly surprising that software reuse has been disappointing in practice, while many hold out great hope for component software.

There is some commonality between the ideas of component and infrastructure. Like components, infrastructure is intended for multiple uses, is typically licensed from another company, is typically used as is, and is typically encapsulated. Infrastructure (as seen by the application developer and operator) is large-grain, whereas components are typically smaller-grain. The primary distinction between the components and infrastructure lies in composability. Infrastructure is intended to be extended by adding applications. This is a weak form of composability, because it requires that the infrastructure precede the applications, and the applications are developed specifically to match the capabilities of an existing infrastructure. Similarly, an application is typically designed for a specific infrastructure context and thus lacks the non-context-specific property. Components, on the other hand, embody a strong form of composability in that the goal is to enable the composability of two (or more) components that are developed completely independently, neither with prior knowledge of the other. Achieving this is a difficult technical challenge.

7.3.3 Component Portability

The issue of portability arises with components as well as other software (see figure 7.10). A portable component can be deployed and installed on more than one platform. But it is also possible for distributed components to compose even though they are executing on different platforms. These are essentially independent prop-

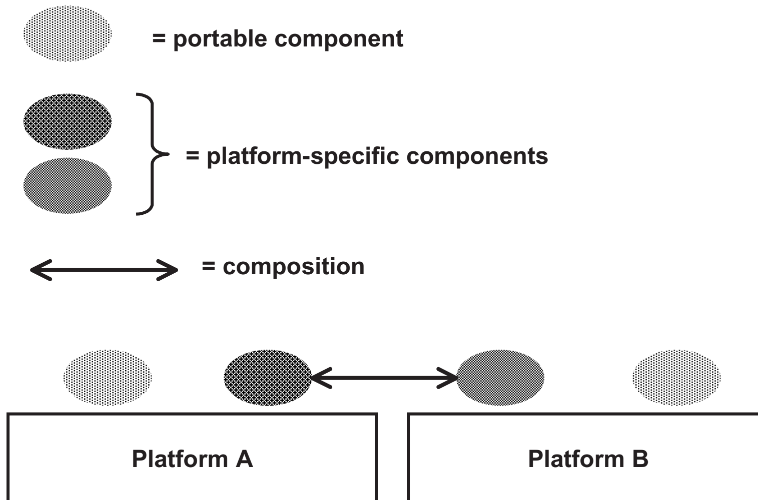

Figure 7.10
Components can be portable, or they can compose across platforms, or both.

erties; a component technology can support one or the other or both of these properties, often limited to specific platforms and environments. Almost all component technologies today support distributed components that execute on the same type of platform.

Insisting on unconditional portability undesirably limits innovation in platforms (see section 4.4.3) but also desirably increases the market for a portable component or eliminates the cost of developing variants for different platforms. Cross-platform distributed component composition offers the advantages of portability (components can participate in applications predominately executing on different platforms) without its limitations (each component can take full advantage of platform-specific capabilities). Here the trade-off is one of efficiency: performance suffers when crossing software and hardware platform boundaries. Of course, this issue also interacts with provisioning and operation (see section 7.3.7).

7.3.4 Component Evolution

There is a fundamental tension in the component methodology relating to the inherent evolution of user requirements (Lehman and Ramil 2000). Recall the distinction between specification-driven and satisfaction-driven software (see section 3.2.11). Particularly if a component is to be licensed rather than developed, it is simpler contractually to view it as specification-driven. It can then be objectively

evaluated by laboratory testing, and the contractual relationship between licensee and licensor is easier to formulate in terms of objective acceptance criteria. The tension comes from the reality that applications as a whole are decidedly satisfaction-driven programs. Can such applications be assembled wholly from specification-driven components? Likely not. In practice, some components (especially those destined for applications) may be judged by satisfaction-driven criteria, greatly complicating issues surrounding maintenance, upgrade, and acceptance criteria. Worse, a successful component is assembled into multiple satisfaction-driven programs, each with distinct stakeholders and potentially distinct or even incompatible criteria for satisfaction of those stakeholders. This tension may limit the range of applicability of components, or reduce the satisfaction with componentized systems, or both. It tends to encourage small-grain components, which are less susceptible to these problems but also incur greater performance overhead. It may also encourage the development of multiple variants on a component, which undercuts important advantages.

A related problem is that a project that relies on available components may limit the available functionality or the opportunity to differentiate from competitors, who have the same components available. This can be mitigated by the ability to mix components with handcrafted or reusable modules and by incorporating a broad set of configuration options into a component.

7.3.5 An Industrial Revolution of Software?

Software components are reminiscent of the Industrial Revolution's innovation of standard reusable parts. In this sense, components can yield an industrial revolution in software, shifting the emphasis from handcrafting to assembly in the development of new software, especially applications. This is especially compelling as a way to reduce the time and cost of developing applications, much needed in light of the increasing specialization and diversity of applications (see section 3.1). It may even be feasible to enable end-users to assemble their own applications. This industrial revolution is unlikely to occur precipitously, but fortunately this is not a case of "all or nothing" because developed or acquired modules can be mixed.

This picture of a software industrial revolution is imperfect. For one thing, the analogy between a software program and a material product is flawed (Szyperski 2002a). If a program were like a material product or machine, it would consist of a predefined set of modules (analogous to the parts of a machine) interacting to achieve a higher purpose (like the interworking of parts in a machine). This was, in fact, the view implied in our discussion of architecture (see section 4.3), but in

practice it is oversimplified. While software is composed from a set of interacting modules, many aspects of the dynamic configuration of an application's architecture are determined at the time of execution, not at the time the software is created. During execution, a large set of modules is created dynamically and opportunistically based on specific needs that can be identified only at that time.

Example A word processor creates many modules (often literally millions) at execution time tied to the specific content of the document being processed. For example, each individual drawing in a document, and indeed each individual element from which that drawing is composed (lines, circles, labels) is associated with a software module created specifically to manage that element. The implementers provide the set of *available* kinds of modules, and also specify a detailed plan by which modules are created dynamically at execution time[12] and interact to achieve higher purposes.

Implementing a modern software program is analogous not to a static configuration of interacting parts but to creating a plan for a very flexible factory in the industrial economy. At the time of execution, programs are universal factories that, by following specific plans, manufacture a wide variety of immaterial artifacts on demand and then compose them to achieve higher purposes. Therefore, in its manner of production, a program—the product of development—is not comparable to a hardware product but is more like a very flexible factory for hardware components. The supply of raw materials of such a factory corresponds to the reusable resources of information technology: instruction cycles, storage capacity, and communication bandwidth. The units of production in this factory are dynamically assembled modules dynamically derived from modules originally handcrafted or licensed as components.

There is a widespread belief that software engineering is an immature discipline that has yet to catch up with more mature engineering disciplines, because there remains such a deep reliance on handcrafting as a means of production. In light of the nature of software, this belief is exaggerated. Other engineering disciplines struggle similarly when aiming to systematically create new factories, especially flexible ones (Upton 1992). Indeed, other engineering disciplines, when faced with the problem of creating such factories, sometimes look to software engineering for insights. It is an inherently difficult problem, one unlikely to yield to simple solutions. Nevertheless, progress will be made, and the market for components will expand.

A second obstacle to achieving an industrial revolution of software is the availability of a rich and varied set of components for licensing. It was argued earlier

that purchasing components in a marketplace is more promising than using internally developed components because the former offers higher scale and significant economic benefit to the developer/supplier. Such a *component marketplace* is beginning to come together.

Example Component technologies are emerging based on Microsoft Windows (COM+ and CLR) and Sun Microsystems' Java (JavaBeans and Enterprise JavaBeans). Several fast-growing markets now exist (Garone and Cusack 1999), and a number of companies have formed to fill the need for merchant, broker, and triage roles, including ComponentSource and FlashLine. These firms provide an online marketplace where buyers and sellers of components can come together.

One recognized benefit of the Industrial Revolution was the containment of complexity. By separating parts suppliers and offering each a larger market, economic incentives encouraged suppliers to go to great lengths to make their components easier to use by abstracting interfaces, hiding and encapsulating the complexities. New components will also tend to use existing standardized interfaces where feasible rather than creating new ones (so as to maximize the market), reducing the proliferation of interfaces. Thus, a component marketplace may ultimately be of considerable help in containing software complexity, as it has for material goods and services.

Another obstacle to an industrial revolution of software, one that is largely unresolved, is trust and risk management. When software is assembled from components purchased from external suppliers, warranty and insurance models are required to mitigate the risk of exposure and liability. Because of the complexity and characteristics of software, traditional warranty, liability laws, and insurance require rethinking in the context of the software industry, an issue as important as the technical challenges.

Another interesting parallel to component software may be biological evolution, which can be modeled as a set of integrative levels (such as molecules, cells, organisms, and families) where self-contained entities at each level (except the bottom) consist mainly of innovative composition of entities from the level below (Petterersson 1996).[13] Like new business relationships in an industrial economy, nature seems to evolve ever more complex entities in part by this innovative composition of existing entities, and optimistically components may unleash a similar wave of innovation in software technology.

It should be emphasized that these parallels to the industrial economy and to biological evolution depend upon the behavioral nature of software (which distin-

guishes it from the passive nature of information), leading directly to the emergence of new behaviors through module composition.

7.3.6 Component Standards and Technology

Assembling components designed and developed largely independently requires standardization of ways for components to interact. This addresses interoperability, although not necessarily the complementarity also required for the composition of components (see section 4.3.6). Achieving complementary, which is more context-specific, is more difficult. Complementarily is addressed at the level of standardization through bodies that form domain-specific reference models and, building on those, reference architectures. Reference architectures devise a standard way to divide and conquer a particular problem domain, predefining roles that contributing technologies can play. Components that cover such specified roles are then naturally complementary.

Example The Object Management Group, an industrial standardization consortium, maintains many task force groups that establish reference models and architectures for domains such as manufacturing, the health industry, or the natural sciences.

Reusability or multiple use can focus on two levels of design: architecture and individual modules. In software, a multiuse architecture is called a *reference architecture*, a multiuse architecture cast to use specific technology is called a *framework*, and a multiuse module is called a component. In all cases, the target of use is typically a narrowed range of applications, not all applications. One reason is that, in practice, both the architecture and the complementarity required for component composition requires some narrowing of application domain. In contrast, infrastructure software targets multiple-use opportunities for a wide range of applications.

Example Enterprise resource planning (ERP) is a class of application that targets standard business processes in large corporations. Vendors of ERP, such as SAP, Baan, Peoplesoft, and Oracle, use a framework and component methodology to provide some flexibility to meet varying end-user needs. Organizations can choose a subset of available components, and mix and match components within an overall framework defined by the supplier. (In this particular case, the customization process tends to be so complex that it is commonly outsourced to business consultants.)

The closest analogy to a framework in the physical world is called a platform (leading to possible confusion, since that term is used differently in software; see section 4.4.2).

Example An automobile platform is a standardized architecture and associated components and manufacturing processes that can be used as the basis of multiple products. Those products are designed by customizing elements fitting into the architecture, like the external sheet metal.

In essence a framework is a preliminary plan for the decomposition of (parts of) an application, including interface specifications. A framework can be customized by substituting different functionality in constituent modules and extended by adding additional modules through defined gateways. As discussed in section 7.3.5, a framework may be highly dynamic, subject to wide variations in configuration at execution time. The application scope of a framework is necessarily limited: no single architecture will be suitable for a wide range of applications. Thus, frameworks typically address either a narrower application domain or a particular vertical industry.

Component methodologies require discipline and a supporting infrastructure to be successful. Some earlier software methodologies have touted similar advantages but have incorporated insufficient discipline.

Example Object-oriented programming is a widely used methodology that emphasizes modularity, with supporting languages and tools that enforce and aid modularity (e.g., by enforcing encapsulation). While OOP does result in an increase in software reuse in development organizations, it has proved largely unable to achieve component assembly. The discipline as enforced by compilers and runtime environment is insufficient, standardization to enable interoperability is inadequate, and the supporting infrastructure is also inadequate.

From a development perspective, component assembly is quite distinctive. Instead of viewing source code as a collection of textual artifacts, it is viewed as a collection of units that separately yield components. Instead of arbitrarily modifying and evolving an ever-growing source base, components are individually and independently evolved (often by outside suppliers) and then composed into a multitude of software programs. Instead of assuming an environment with many other specific modules present, a component provides documented connection points that allow it to be configured for a particular context. Components can retain their separate identity in a deployed program, allowing that program to be updated and extended by replacing or adding components. In contrast, other programming methodologies deploy a collection of executable or dynamically loadable modules whose configuration and context details are hard-wired and cannot be updated without being replaced as a monolithic whole.

Even an ideal component will depend on a platform's providing an execution model to build on. The development of a component marketplace depends on the availability of one or more standard platforms to support component software development, each such platform providing a largely independent market to support its component suppliers.

Example There are currently two primary platforms for component software, the Java universe, including Java 2 Enterprise Edition (J2EE) from Sun Microsystems, and the Windows .NET universe from Microsoft. Figure 7.11 illustrates the general architecture of Enterprise JavaBeans (EJB), part of J2EE, based on the client-server model. The infrastructure foundation is a set of (possibly distributed) servers and (typically) a much larger set of clients, both of which can be based on heterogeneous underlying platforms. EJB then adds a middleware layer to the servers, called an *application server*. This layer is actually a mixture of two ideas discussed earlier. First, it is value-added infrastructure that provides a number of common services for many applications. These include directory and naming services, database access, remote component interaction, and messaging services. Second, it provides an environment for components, which are called Beans in EJB. One key aspect of this environment is the component container illustrated in figure 7.12. All interactions

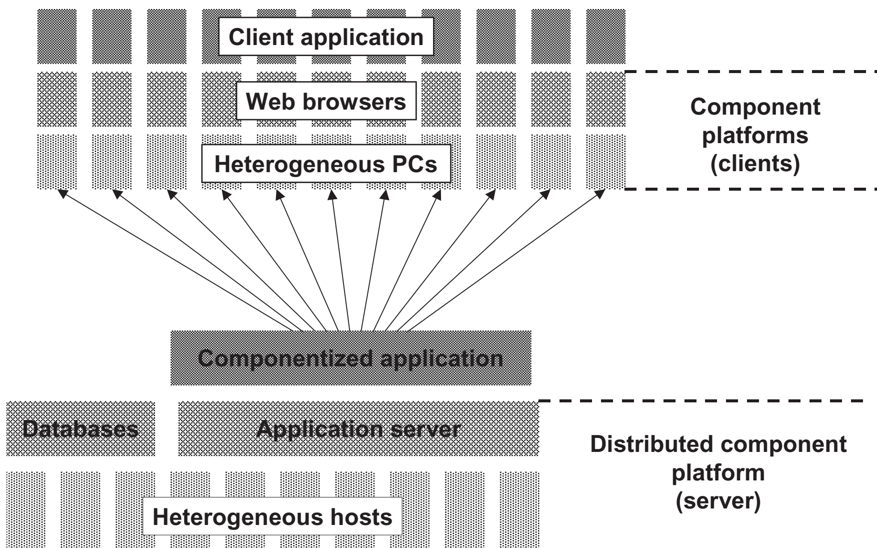

Figure 7.11
An architecture for client-server applications supporting a component methodology.

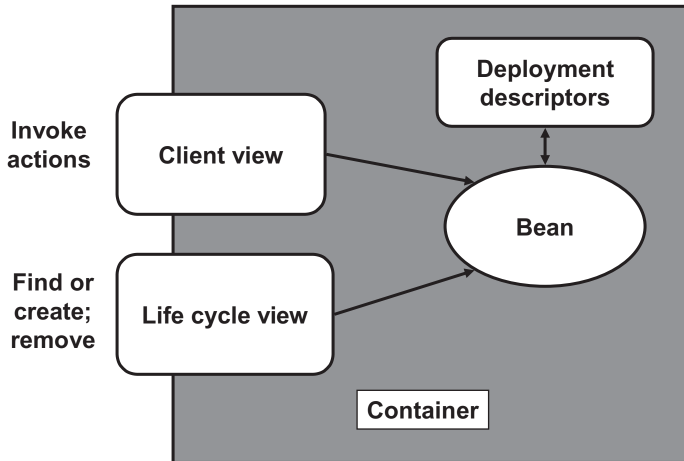

Figure 7.12
A component container in JavaBeans.

with a given component are actually directed at its container, which in turn interacts with the component itself. This allows many additional value-added services to be transparently provided on behalf of the component, such as supporting multiple clients sharing a single component, state management, life cycle management, and security. The deployment descriptors allow components to be highly configurable, with their characteristics changeable according to the specific context. In addition, J2EE provides an environment for other Java components in the Web server (servlets) and client Web browser (applets) based on a Java browser plug-in, as well as for client-side applications. In the .NET architecture, the role of application server and EJBs is played by COM+ and the .NET Framework. The role of Web server and Web service host is played by ASP.NET and the .NET Framework. Clients are covered by client-side applications, including Web browsers with their extensions.

Component-based architectures should be modular, particularly with respect to weak coupling of components, which eases their independent development and composability (see section 4.3.3). Strong cohesion within components is less important because components can themselves be hierarchically decomposed for purposes of implementation. Market forces often intervene to influence the granularity of components, and in particular sometimes encourage course-grain components with considerable functionality bundled in to reduce the burden on component users and to help encapsulate implementation details and preserve trade secrets (see chapter 8).

In some cases, it is sheer performance objectives that encourage course-grained components because there is an unavoidable overhead involved in interacting with other components through fixed interfaces.

Constructing an application from components by configuring them all against one another, called a *peer-to-peer architecture*, does not scale beyond simple configurations because of the combinatorial explosion of created dependencies, all of which may need to be managed during the application evolution. A framework can be used to bundle all relevant component connections and partial configurations, hierarchically creating a coarser-grain module. A component may plug into multiple places in a component framework, if that component is relevant to multiple aspects of the system. Figure 7.13 illustrates how a framework can decouple disjoint dimensions.[14] This is similar to the argument for layering (see figure 7.6), standards (see section 7.2), and commercial intermediaries, all of which are in part measures to prevent a similar combinatorial explosion.

Example An operating system is a framework, accepting device driver modules that enable progress below (although these are not components because they can't compose with one another) and accepting application components to enable progress above (assuming that applications allow composability). Allowing component frameworks to be components themselves creates a component hierarchy.

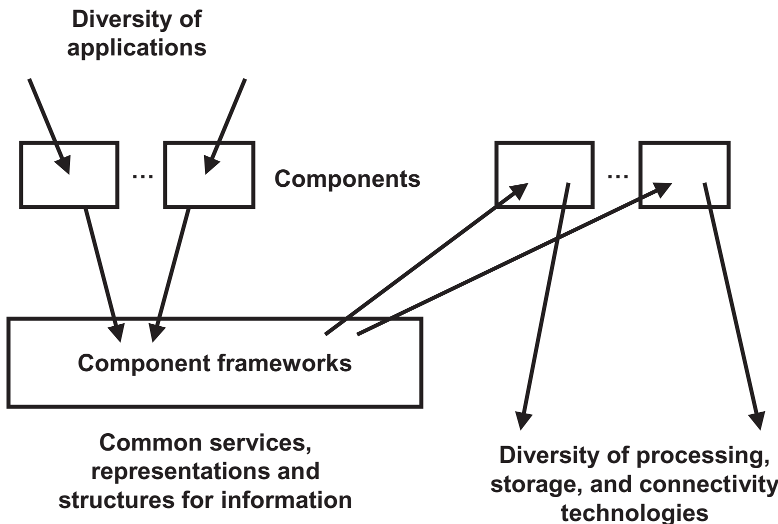

Figure 7.13
Component frameworks to separate dimensions of evolution.

For example, an OS-hosted application can be turned into a framework by accepting add-ins (a synonym for components).

7.3.7 Web Services

The provisioning and operation of an entire application can be outsourced to a service provider (see section 6.2). Just as an application can be assembled from components, so too can an application be assembled from the offerings of not one but two or more service providers. Another possibility is to allow end-users to assemble applications on a peer-to-peer basis by offering services to one another directly, perhaps mixed with services provided by third-party providers. This is the idea behind *Web services*, in which services from various service providers can be considered components to be assembled. The only essential technical difference is that these components interact over the network; among other things, this strongly enforces the encapsulation property. Of course, each component Web service can internally be hierarchically decomposed from other software components. Services do differ from components significantly in operational and business terms in that a service is backed by a service provider who provisions and operates it (Szyperski 2001), leading to different classes of legal contracts and notions of recompense (Szyperski 2002b). The idea that components can be opportunistically composed even after deployment is consistent with the Web service idea.

Distributed component composition was considered from the perspective of features and functionality (see section 7.3.3). Web services address the same issue from the perspective of provisioning and operation. They allow component composition across heterogeneous platforms, with the additional advantage that deployment, installation, and operation for specific components can be outsourced to a service provider. They also offer the component supplier the option of selling components as a service rather than licensing the software. Web services shift competition in the application service provider model from the level of applications to components.

If the features and capabilities of a specific component are needed in an application, there are three options. The component can be deployed and installed on the same platform (perhaps even the same host) as the application. If the component is not available for that platform (it is not portable), it can be deployed and installed on a different host with the appropriate platform. Finally, it can be composed into the application as a Web service, avoiding the deployment, installation, and operation entirely but instead taking an ongoing dependence on the Web service provider.

A Web service architecture can be hierarchical, in which one Web service can utilize services from others. In this case, the "customer" of one service is another piece of software rather than the user directly. Alternatively, one Web service can be considered an intermediary between the user and another Web service, adding value by customization, aggregation, filtering, consolidation, and similar functions.

Example A large digital library could be assembled from a collection of independently managed specialized digital libraries using Web services (Gardner 2001). Each library would provide its own library services, such as searching, authentication and access control, payments, copy request, and format translations. When a user accesses its home digital library and requests a document not resident there, the library could automatically discover other library services and search them, eventually requesting a document copy in a prescribed format wherever it is found and passing it to the user.

An important element of Web services is the dynamic and flexible means of assembling different services without prior arrangement or planning, as illustrated in this example. Each service can advertise its existence and capabilities so that it can be automatically discovered and those capabilities invoked. It is not realistic, given the current state of technology, to discover an entirely new type of service and automatically interact with it in a complex manner. Thus, Web services currently focus on enabling different implementations of previously known services to be discovered and accessed (Gardner 2001).

Web services could be built on a middleware platform similar to the application server in figure 7.11, where the platform supports the interoperability of components across service providers. However, this adds another level of coordination, requiring the choice of a common platform or alternatively the interoperability of different platforms. A different solution is emerging, standards for the representation of content-rich messages across platforms.

Example XML (see section 7.1.5) is emerging as a common standardized representation of business documents on the Web, one in which the meaning of the document can be extracted automatically. The interoperability of Web services can be realized without detailed plumbing standards by basing it on the passage of XML-represented messages. This high-overhead approach is suitable for interactions where there are relatively few messages, each containing considerable information.[15] It also requires additional standardization for the meaning of messages expressed in XML within different vertical industry segments. For example, there will be a standard for expressing invoices and another standard for expressing digital library search and copy requests.

The dynamic discovery and assembling of services are much more valuable if they are vendor- and technology-neutral. Assembling networked services rather than software components is a useful step in this direction, since services implemented on different platforms can be made to appear identical across the network, assuming the appropriate standards are in place.

Example Recognizing the value of platform-neutral standards for Web services, several major vendors hoping to participate in the Web services market (including Hewlett-Packard, IBM, Microsoft, Oracle, and Sun) work together to choose a common set of standards. SOAP (simple object access protocol) is an interoperability standard for exchanging XML messages with a Web service, UDDI (universal description, discovery, and integration) specifies a distributed registry or catalog of available Web services, and WSDL (Web services description language) can describe a particular Web service (actions and parameters). Each supplier is developing a complete Web services environment (interoperable with others because of common standards) that includes a platform for Web services and a set of development tools.[16] UDDI and Web services work together as illustrated in figure 7.14,

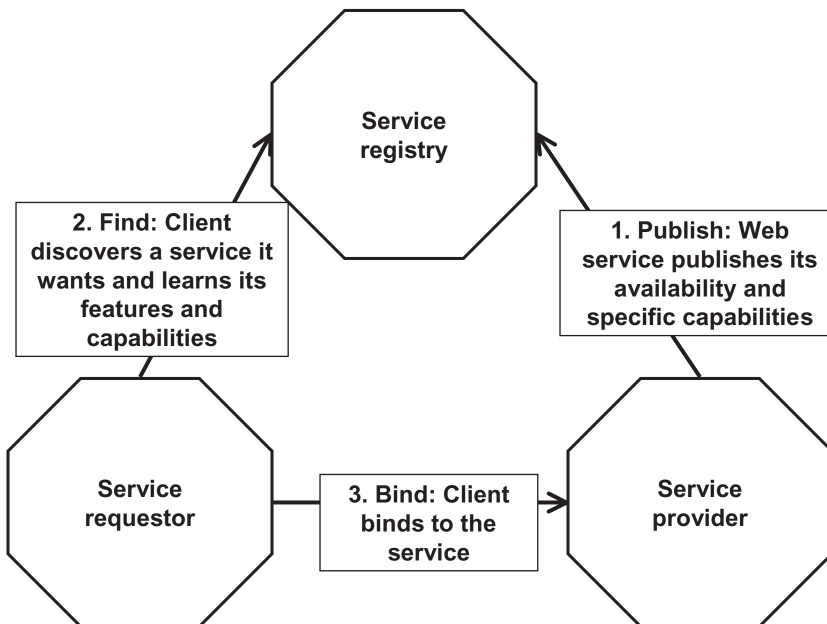

Figure 7.14
The Web services reference model, now on its way toward adoption as a standard, shows how UDDI supports service discovery (Gardner 2001).

with three primitive operations: publish, find, and bind. First, each service provider advertises the availability of its Web services in a common service registry. A service requestor discovers a needed service in the registry, connects to the service (including functions like authentication and arranging for payment), and finally starts using it.

7.4 Research and Discussion Issues

1. One recurring theme is the indirect network effect of applications and infrastructure: applications seek widely deployed infrastructure, and infrastructure seeks a diversity of applications. What are some specific ways in which the market can overcome this chicken-and-egg conundrum? Can you give examples of these? What are their relative merits? Which are most likely to be commercially viable?

2. Consider in specific terms what properties an infrastructure must have to enable new applications to arise without the knowledge or participation of the infrastructure supplier or service provider.

3. Give some additional examples of how the tendency of the market to add to infrastructure rather than change it leads to later difficulties. Can you identify some examples and specific methodologies in which infrastructure can be changed rather than merely expanded?

4. What are the best areas of the software industry for venture-funded companies? for large established companies? Why?

5. Because of the substantial economies of scale, it is wise to avoid direct undifferentiated competition in the software industry. What are some specific strategies that suppliers use to differentiate themselves in applications? in infrastructure?

6. A dogmatic view of layering requires that layers be homogeneous and interact only with the layers directly above and below. What are some advantages and disadvantages of relaxing this dogma, as in the example of figure 7.3?

7. What are some differences in the strategic challenges of firms competing in a stovepipe and in a layered structure in the industry?

8. Consider how the properties of good software modularity (see section 4.3.3) apply more or less strongly to industrial organization.

9. Discuss in some depth whether an infrastructure service provider should also bundle applications, and whether it should develop or acquire those applications.

10. Discuss in some depth the relation between standardization and industrial organization, specifically different types of standardization (e.g., de facto vs. an industry standards body).

11. What strategies do software suppliers use to maintain control over an architecture and in taking advantage of that architectural control?

12. What are the considerations in choosing to offer an API or not?

13. What are all the elements that make up an open standard, and when these elements are modified or eliminated, when does the standard cease to be open? This question is best considered in the context of what kind of benefits an open standard confers, and when those benefits are compromised.

14. Consider the process by which a de facto standard develops in the marketplace. Is a de facto standard always an open standard? When is encouragement of a de facto standard in the best interest of a supplier? Do different types of de facto standards (e.g., interfaces, APIs, data representations) arise in similar ways?

15. Discuss the relation between an agile software development process (see section 4.2.5) and an agile standardization process (one that is flexible and adaptive). Under what conditions is a standardization process essentially (or at least part of) a collective development process?

16. Composability of modules requires interoperability and complementarity (see section 4.3.6). What is the proper role of standardization in these two aspects? What else is required besides standardization?

17. Consider mechanisms and incentives that could be used to encourage software reuse in a development organization. Do you think this can be effective? What disadvantages might there be?

18. For each property of a component listed in table 7.5, discuss the business considerations and possibilities that arise from that property. What are the implications to component supplier, component customer (e.g., system integrator or system developer), and end-user?

19. Discuss how component software differs from infrastructure software, since both concepts emphasize multiple uses. How do the challenges and strategies of component suppliers differ from those of application and infrastructure suppliers?

20. Reflect further on the challenges of satisfaction-based requirements for a component supplier-customer relationship. Also consider the issues of risk and trust, and how these relate to satisfaction.

21. Imagine that you are in the business of producing and selling plans for a flexible factory. What kind of business challenges would you face?

22. Discuss the possible parallels and differences between component assembly as a model of emergence in software and business relationships as a model of emergence in the industrial economy. What can be learned from these parallels?

23. Repeat the last question, substituting innovative composition of entities as a model for emergence in biological evolution.

24. How does the philosophy of Web services (assembling applications from component services) differ from that of the application service provider (accessing fully integrated applications over a wide-area network)? What are the differences in terms of business models, opportunities, and challenges?

25. Discuss the business issues that surround the assembly of applications from two or more Web services. These include the necessary coordination and issues of trust and responsibility that arise from outsourcing component services.

7.5 Further Reading

Mowery and Nelson (1999), Mowery and Langlois (1994), and Torrisi (1998) describe the general structure of the software industry, particularly from a historic perspective. Ferguson and Morris (1994) give a useful introduction to the computer industry, with an emphasis on hardware and the importance of architecture. While there are a number of books on software reuse, the Jacobson, Griss, and Jönsson (1997) book is one of the best. Szyperski (2002a) has the most comprehensive treatment of software components, including both economics and business issues. The state-of-the-art collection of edited articles in Heineman and Councill (2001) provides a wealth of additional information on most related issues.