T H R E E

Software Architecture: Basic Training

This chapter on basic training for software architects presents the fundamental tools required of effective software architects. In the military, basic training is used to challenge and motivate cadets and to demonstrate both the demands and rewards of a military career. Similarly, software architects must be motivated individuals who have the desire to confront the challenges of technical leadership in a software development effort. However, motivation is not enough. A software architect must be equipped with the intellectual tools to realize an architectural vision.

This book takes a hands-on approach that not only presents the best architectural practices in the industry but also provides concrete real-world examples and exercises for applying the presented material to circumstances common throughout the software industry. Basic training will cover the fundamental concepts of software technology, which provide a foundation for software architecture. Software technology has evolved through many trends and alternatives for software development. Currently, mainstream software practice has evolved from procedural to object-oriented to component-oriented development (Figure 3.1). With the increasing adoption of enterprise Java and Microsoft .Net, component orientation is the next major paradigm. In corporate development, most new-start projects are adopting component orientation because it is supported by the majority of commercial development environments. As is discussed in this chapter, object orientation has a very weak notion of software architecture, which leads to serious shortcomings. The emerging trend of component orientation is replacing old approaches with strong elements of architectural design.

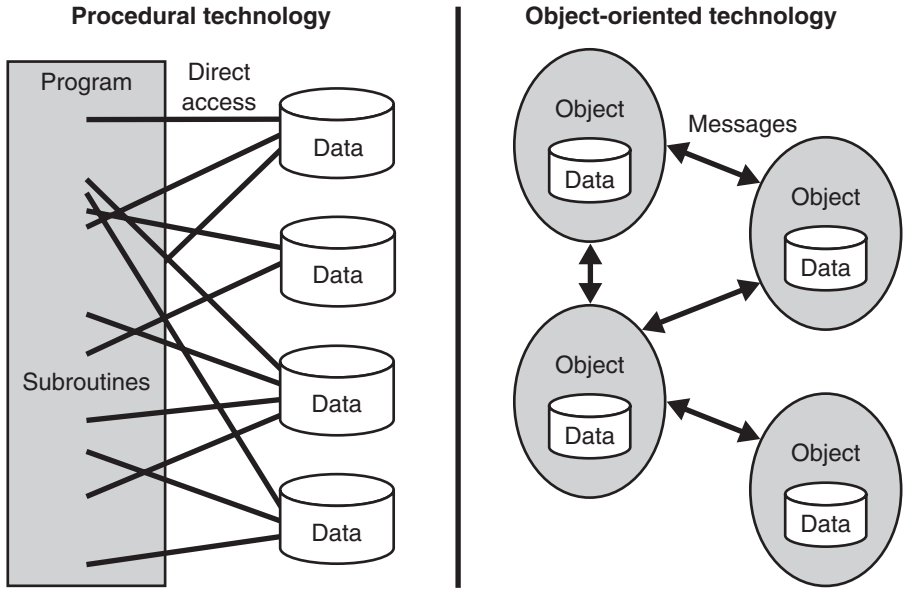

FIGURE 3.1 (a) Procedural Paradigm and (b) Object-Oriented Paradigm

Software architects must be able to articulate these development paradigms clearly, along with appropriate uses of enabling technologies. In any given project, an eclectic mixture of development paradigms (including relational database management) can be useful to achieve the best results. Each paradigm has something useful to offer, including mature development tools.

Today, most organizations will find their technology skill base engaged in one of the three major paradigms: procedural, object oriented, or component oriented. Each paradigm is highly specific to the organization and its staff skills. Procedural and object-oriented paradigms are closely tied to programming language choice, but component orientation is different in that it is more closely associated with the selection of an infrastructure.

Procedural programming languages include FORTRAN, COBOL, C, Pascal, and BASIC. In procedural technology, the program comprises the process for executing various algorithms. The process is separated from the data in the system, and the process manipulates the data through direct access operations. This is a direct outcome of the stored-procedure programming systems from which computer technology originates. When the program and data are separated, there are many potential interdependencies between parts of the program. If the data representation is modified, there can be substantial impacts on the program in multiple places.

An example of data–process separation is the year 2000 problem, in which simply adding some additional digits to the date representation had catastrophic consequences for procedural software. Unfortunately, because the majority of systems are built with procedural technology, the dependencies upon these data representations can cause system-wide program errors and the necessity for line-by-line program review and modification.

3.1 OBJECT-ORIENTED TECHNOLOGY

Object-oriented programming languages include Smalltalk, C++, the Java programming language ("the Java language"), and C#, one of the languages available in the Microsoft .Net development environment. These languages support the encapsulation of data with accessor code in terms of abstract data types (commonly called classes). In object-oriented programming languages, the encapsulation capabilities are sufficient for reasonably sized programs. As long as software modules are maintained by individual programmers, encapsulation is sufficiently robust to provide some intrinsic benefits. However, language-specific encapsulation is insufficient to support software reuse and distributed systems.

In object-oriented technology, the basic paradigm is changed to enable a separation of concerns. Figure 3.1 shows the object-oriented technology paradigm in which the program is broken up into smaller pieces called *objects.* Each object contains some of the data of the system, and the program encapsulates that data. In other words, access to the data is available only by using the program through which it is directly associated. In this way, the system is partitioned into modules that isolate changes. Any changes in data representation have an impact only on the immediate object that encapsulates that data.

Objects communicate with each other through messages. Messages can have an impact upon state—in other words, changing the data—but only through the encapsulated procedures that have an intimate relationship to the local data. For small-scale programs, the object paradigm is effective in its isolation of change. However, the paradigm is not perfect for all of its potential uses.

Object-oriented technology is in widespread use today. It has been said that the procedural technologies originated from academia, but the object-oriented technologies actually originated from commercial organizations. In fact, object-oriented technologies have many interesting origins that go back virtually to the beginning of computer science. Today, object technology is the

dominant paradigm for commercial software. Virtually every vendor in the software business is providing object-technology solutions that, together with component infrastructures, can enable interoperability between software vendors in various software environments.

OBJECT-ORIENTED ARCHITECTURE

For the majority of practitioners, object orientation is devoid of a software architecture approach. This is manifested in multiple ways in object-oriented methods and culture. Starting with what is generally regarded as the original source for OO thinking, *Designing Object-Oriented Software* [Wirfs-Brock 1990], there was a notion of software architecture, including the discovery of subsystems through inspection of collaboration diagrams, which merited an entire chapter. In the next decade, little was written about architecture in the OO methodology community. Major OO methodology books had at most a few paragraphs concerning architecture, which were a faint reflection of Wirfs-Brock's architectural notions.

Since virtually nothing was written about architecture in the literature, most OO practitioners had only rudimentary architectural guidance. They had no reason to consider architecture important. This attitude has led to great confusion on OO projects, as team members struggle to manage complexity and scalability with OO methods not designed to address them.

In general, OO methods involve successive refinement of object models, where most analysis objects are eventually transformed into programming objects. In terminology used in this book, these approaches are called model-based methods. The assumption that each analysis object will inevitably become a programming object is a major obstacle for OO thinkers to overcome in order to understand architecture. In architectural models, specification objects represent constraints, not programming objects. They may or may not be transformed into programming objects; that is an independent decision on the part of the developer.

OO also opposes architecture in other subtle ways, related to project culture. OO encourages project teams to be egalitarian (e.g., CRC cards), where all decisions are democratic. On such a project, there is no role for the architect because there is little separation of decision making between members of the development team.

OO encouraged "bad-is-better" thinking in development, a philosophy that is virtually the opposite of architectural thinking. Using bad-is-better

thinking, the external appearance of a working implementation greatly outweighs any requirement for maintainable internal structure. In other words, rapid iterative prototyping, with ruthless disregard for architectural principles, is a normal, healthy environment for OO development.

The topic of architecture has resurfaced only recently in OO literature, with the new-found popularity of componentware. Now it is customary to include a token chapter on architecture in most methodology books, whereas in the heyday of OO, architecture was virtually taboo. In one sense, componentware is a response to the shortcomings of OO. Componentware, with its emphasis on larger-grained software interfaces and modules, is a progressive step toward an architectural mindset.

Databases and Objects

Database technologies are also evolving toward objects. The database technologies originated with several different models. In recent years, the relational model of databases has been predominant. More recently, object-oriented databases have become a significant technology market, and databases that combine object orientation and relational concepts are commonplace. Most of the major industry databases, such as Oracle 9i and IBM's DB2 database, include object-relational capabilities.

Database query languages, such as Structured Query Language (SQL), are being extended in standards work to support object-oriented concepts. One reason why this is occurring is that the kinds of applications people are creating require substantially more sophisticated types of data representations and types of query algorithms for searching and manipulating the information.

Object in the Mainstream

Object technology is used today in most application areas and vertical markets. Government organizations and commercial industry are pursuing dozens of projects in object technology. A principal advantage of technology is that it enables the implementation of new business processes that provide competitive advantage to organizations. Society is changing toward increasing dependence upon information technology. The use of object technology enables rapid system implementation and various forms of labor saving through software reuse mechanisms. Even though the largest number of lines of software still exists in procedural languages such as COBOL, it is becoming clear that this paradigm is changing.

Scripting Languages

Proponents of scripting languages claim that there are more scripting language programmers than any other kind [Ousterhout 1998]. Scripting languages such as the JavaScript language, TCL shell programming languages, and Visual Basic enable preexisting software (e.g., components) to be easily integrated into application configurations.

3.2 COMPONENT-ORIENTED TECHNOLOGY

Moving to the next level of software sophistication requires fundamental changes in systems thinking, software processes, and technology utilization. The next major area of technology, componentware (or component orientation), contains key elements of the solution to today's critical software problems.

The componentware approach introduces a set of closely interrelated techniques and technologies. Componentware introduces a sophisticated mindset for generating business results. These componentware elements include

- Component infrastructures
- Software patterns
- Software architecture
- Component-based development

Componentware technologies provide sophisticated approaches to software development that challenge outdated assumptions. Together these elements create a major new technology trend. Componentware represents as fundamental a change in technology as object orientation did in previous generations. These componentware technologies are discussed after a brief introduction to componentware's unique principles.

Components versus Objects

Componentware can be understood as a reincarnation of object orientation and other software technologies. Distinguishing componentware from previous generations of technology are four principles: encapsulation, polymorphism, late binding, and safety. This list overlaps with object orientation, except that it eliminates the emphasis on inheritance. In component thinking, inheritance is a tightly coupled, white-box relationship that is unsuitable for most forms of packaging and reuse. Instead, components reuse functionality by invoking

other objects and components instead of inheriting from them. In component terminology, these invocations are called *delegations.*

> *"Fit the parts together, one into the other, and build your figure like a carpenter builds a house. Everything must be constructed, composed of parts that make a whole..."*—Henri Matisse

By convention, all components have specifications corresponding to their implementations. The specification defines the component *encapsulation* (i.e., its public interfaces to other components). Reuse of component specifications is a form of *polymorphism* that is strongly encouraged. Ideally, component specifications are local or global standards that are widely reused throughout a system, an enterprise, or an industry.

Componentware utilizes composition for building systems. In composition, two or more components are integrated to create a larger entity, which could be a new component, a component framework, or an entire system. Composition is the integration of components. The combined component acquires joint specifications from the constituent component.

If the components have matching specifications for client calls and services, then they can interoperate with no extra coding. This is often called plug-and-play integration. When executed at runtime, this is a form of *late binding*. For example, a client component can discover a component server through an online directory, such as the CORBA Trader Service. With matching client and service interface specifications, the components can establish a runtime binding to each other and interact seamlessly through the component infrastructure.

In a perfect world, all components would be fully conformant with their specifications and free from all defects. Successful operation and interoperation of components depend on many internal and external factors. *Safety* properties can help because they can minimize entire classes of defects in a component environment. As society becomes increasingly dependent upon software technology, safety has become a serious legal concern and one of the most important areas of computer science research. For example, Java's garbage collection feature guarantees memory safety, or freedom from memory deallocation defects (which are problematic in C++ programs). Other kinds of safety include type safety (guaranteed data type compatibility) and module safety, which controls the effects of software extension and component composition.

Component Infrastructures

The componentware revolution has already arrived in the form of component infrastructures. Major platform vendors have bet their futures on componentware product lines. In particular, Microsoft, Sun Microsystems, IBM, and the CORBA consortia have established significant componentware infrastructures through massive technology and marketing investments.

These component infrastructures (Microsoft .Net and Sun Java Enterprise JavaBeans including CORBA) are dominant infrastructures for competing industry segments—component-oriented enterprise application development. Interestingly, these technologies are also mutually interoperable to a great extent through the mutual support of XML, Web services, and other emerging standards for enterprise application development.

Microsoft has completely revamped its component infrastructure in its current .Net product suite. The Microsoft .Net product suite is focused on the development and deployment of enterprise-level applications and distributed services. While it incorporates a lot of new code, it also includes many of the products and technologies that were successful in Microsoft's previous distributed development platforms, Microsoft Distributed Network Architecture (DNA) and Distributed Common Object Model (DCOM) such as the Microsoft Transactions Server (MTS), Microsoft SQL Server, and the long-lived Common Object Model (COM)/COM+ components. The existing .Net suite upgrades these products in an integrated enterprise product suite along with new support for XML data, Web services, and a much improved development environment. Microsoft .Net is the concrete proof the software industry needed to verify that Microsoft will remain a major player in the emerging world of components for the foreseeable future.

Sun Microsystems' invention of the Java language is a continuing evolution of programming language features, infrastructures, and related class libraries. The Java language technology has created tremendous industry excitement and support from independent developers. The extensions for JavaBeans and Enterprise JavaBeans establish an evolving component model that rivals COM and ActiveX in the cross-platform application space. Enterprise JavaBeans and the IBM San Francisco project are using Java Remote Method Invocation (RMI) for distributed computing, one of several proprietary infrastructures available to Java language programmers. More recently, the Java language has included RMI over OMG's Internet Inter ORB Protocol (IIOP), which allows for Java to interoperate with distributed components in other programming languages that have a CORBA IDL interface. While proprietary Java language infrastructures do provide convenience for programmers, they

require additional complexity to interoperate with other programming languages, especially locally using the Java Native Interface (JNI) to invoke routines in programming languages other than Java. This may be a significant hindrance for corporate projects because it slows legacy integration and cross-language development, which is commonplace for server applications.

Java application servers have overtaken CORBA's role in many Internet-savvy organizations. What CORBA lacks is direct support for scalability, reliability, and maintainability. These capabilities are standard features supported by most Java application servers today.

Componentware infrastructures are having a significant impact on software development. In many respects, these infrastructures are well on their way to becoming mainstream development platforms. Because they are all becoming interoperable (through CORBA IIOP), there is a well-understood relationship between infrastructure models. Their similarities are much greater than their proprietary differences might imply.

Infrastructure selection is one of the most discussed, but least important, aspects of implementing componentware. For corporate developers, the most critical issues are confronted well after infrastructure selection. These issues include how to master designing with the technology, how to architect systems, and how to coordinate one's development efforts..

Component Software Design Patterns

Software design patterns comprise a common body of software knowledge that can be applied across all component infrastructures. *Software design patterns* are proven design approaches to a specific category of structural software problems that are documented for reuse by other developers. Other important categories of patterns include analysis patterns and AntiPatterns. Analysis patterns define proven ways of modeling business information that can be directly applied to the modeling of new software systems and databases.

Software design patterns are a necessary element of componentware. The development of new, reusable components requires an expert-level quality of design, specification, and implementation. Proven design solutions are necessary to establish successful component architectures and frameworks for families of applications. Often, there are too many variables to take chances on unproven design concepts.

The popularity of software design patterns can be explained as a response to the practical shortcomings of object orientation. AntiPatterns explain the common mistakes that people make when developing object-oriented software

systems (as well as other types of systems). Much more is needed than basic object-oriented principles to build successful systems. Design patterns explain the additional, sophisticated ideas that are required for effective software designs. Analysis patterns present the sophisticated ideas necessary for the effective modeling of concepts and data.

It is still commonplace in software development to reinvent design ideas, incurring the risks and delays of trial-and-error experimentation. In fact, most software methods encourage reinvention as the normal mode of development. Given the challenging forces of requirements change, technology innovation, and distributed computing, reinvention is an unnecessary risk in many circumstances. This comment is especially applicable to the development of components, where the costs of defects and redesigns can affect multiple systems.

Altogether, software design patterns can be described as *knowledge reuse*. It is interesting to note that most patterns are considered as simple as common sense by expert-level developers. However, for the majority of developers, patterns are a necessary part of the technical training that can help them to achieve world-class results.

Component Software Architecture

Software architecture concerns the planning and maintenance of system structure from earliest system concept through development and operations. Good architectures are stable system structures that can accommodate changes in requirements and technologies. Good architectures ensure the continuous satisfaction of human needs (i.e., quality) throughout system life cycles. Reusable components are examples of good architecture. They support stable interface specifications, which can accommodate changes that are the result of reuse in many system contexts.

Software architecture plays an important role in component design, specification, and use. Software architecture provides the design context within which components are designed and reused. Components have a role in predetermining aspects of software architecture. For example, a component framework may predefine the architecture of a significant portion of a system.

One of the most exciting aspects of software architecture for componentware is supporting distributed project teams. Software architecture comprises a system specification that enables parallel, independent development of the system or its parts. Proper software architecture defines computational boundaries (i.e., APIs) that divide the system into separate testable subsystems. These subsystems can be outsourced to one or more distributed project teams.

3.3 TECHNOLOGY OWNERSHIP

Because object technology is the dominant commercial paradigm, it is important to understand the major kinds of commercial technologies available for the architecture of software systems. The two major categories include proprietary software and open systems software.

Proprietary Software

Proprietary software is a non-standards-compliant product of a single vendor. That single vendor controls the form and function of the software through many iterations of product releases. When today's systems are built, they are dependent upon commercial software to varying degrees. Commercial software is the primary form of software reuse and in practice is a much more effective form of reuse within individual enterprises.

An economy of scale is one reason why commercial software is a more powerful form of reuse. Large numbers of copies of the software are distributed to customers, and the software can be debugged and quality controlled to a degree that exceeds the in-house development capabilities of even the largest end-user enterprises. When end-user enterprises depend upon proprietary software, they are dependent upon the vendors' continued support for existing capabilities, and architecturally many end-users depend upon future features that the vendors claim will be added to the software. When proprietary software is packaged in the form of a public specification or standard, the specification is usually a direct representation of that single software implementation.

Often, when proprietary specifications are put forward in the public domain, it is unlikely that the proprietary implementation will be modified. This leaves the impression that proprietary software can also be an open system standard, when in fact there is no possibility of modifying the underlying technologies. This phenomenon is especially true when millions of software licenses have been distributed and are running on existing software systems. When proprietary technology is put forward, vendors use unique interpretations of software concepts to describe their products. These interpretations can include fundamental modifications to object-oriented principles.

> *"We shape our buildings; thereafter they shape us."* —Sir Winston Spencer Churchill

The most significant aspect of proprietary technology is the provision of application program interfaces (APIs). The APIs to proprietary software define

the boundary between a proprietary implementation and any value-added application software that either an independent software vendor or the end-user provides to build application systems. As proprietary software technologies evolve through multiple releases, the application program interfaces can change.

New capabilities are continuously added to proprietary software, and this exacerbates the complexity of application program interfaces. In many cases, the complexity of the program interfaces available with proprietary software greatly exceeds the functionality needs of end-user organizations. It then becomes appropriate for the end-user organizations to attempt to manage this complexity in various ways. Complexity-management concepts are covered in several chapters, most explicitly in Chapter 5.

In addition to adding new capabilities to proprietary program interfaces, vendors also on occasion may make obsolete their interfaces in software. Doing so can have a significant maintenance impact upon application software. As proprietary software evolves through multiple releases, it is important for users to continue to upgrade the software to remain in synchronization with the mainstream support activities from the proprietary vendor. When the end-users' systems fall behind more than two cycles, it is often necessary to repurchase and reintegrate completely the commercial software in order to synchronize with the currently released version. Many end-users have found that application program interfaces are almost completely obsolete within a few cycles of product release.

In summary, proprietary software releases and the evolution of the program interfaces become a treadmill for application programmers and independent software vendors to maintain synchronization with available and supported software. There is a conflict of interests between the application users and the proprietary software vendors because the majority of the vendors' profits can be driven by the sale of software upgrades.

Open Systems Software

The other major category of commercial software is open systems technologies (Figure 3.2). An open system technology is fundamentally different from a proprietary technology. In an open system technology, multiple vendors agree to develop a specification that is independent of proprietary implementations. This is the case of most formal standards activities and many consortium standards activities, which are becoming increasingly prevalent. In an open systems technology, the specification governs the behavior of the implementations.

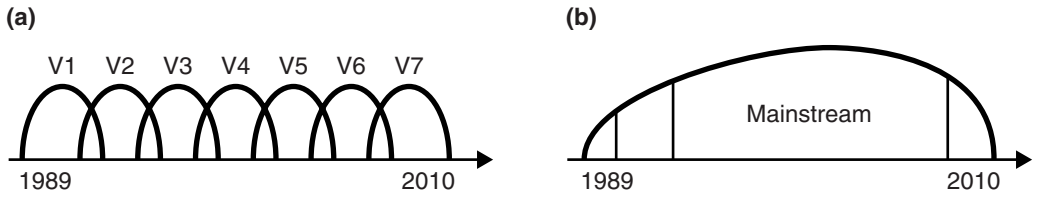

FIGURE 3.2 (a) Proprietary Technology and (b) Open Systems Technology

One of the key benefits is a consistency of implementation interfaces across multiple vendors. Additional benefits include uniformity of terminology and software interfaces because the open systems technology requires multiple vendors to reach consensus. Another benefit is an increased level of technology specification and an extended life cycle. Because product developments are in parallel across multiple vendor organizations, the corresponding marketing activities that create the demand for the technology are also synchronized and coordinated. A key benefit of open systems technology is the interoperability that it provides between commercial software vendors. The distinction between open systems and proprietary technologies is particularly appropriate for object-oriented systems, which are becoming the mainstream of application development, as object technology is already the mainstream of commercial technology.

Commercial information technology is evolving. Additional capabilities that increasingly satisfy application needs are being added and are becoming available through commercial technology. However, there is also a significant amount of reinvention in commercial technology of fundamental capabilities such as operating systems and programming languages.

In some commercial technologies, such as office automation, word processors, and spreadsheets, a continual reorganization of functionality is presented to the end-user without significant extension of capabilities. In the view of many people, the rate of technology evolution on the commercial side is relatively slow in comparison to the growth in needs for competitive application developers. Commercial technology is put forth to satisfy the needs of large numbers of users. The generality of this software exceeds the need of any individual application user. In order to adapt commercial technologies to application needs, there is a requirement for software development and installation which customizes the commercial software to the needs of specific applications (Figure 3.3).

The requirement to customize commercial technology is often called *profiling;* this concept is covered in more detail in Chapter 5. In addition to the profiling software, substantial application-specific software is required to cre-

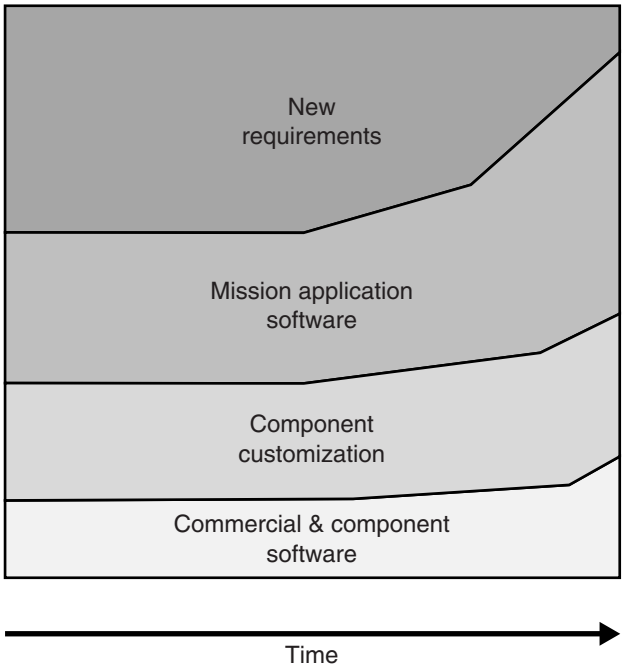

FIGURE 3.3 Commercial Software Customization

ate application systems. Because of the relatively primitive capabilities available commercially for many application needs, this requirement drives an increasing demand to build more and more application-specific software to complete the architecture for application systems. As systems evolve from single-user and departmental-level applications to the enterprise with greater interoperability capabilities, the functional gap between available commercial software and individual user software will continue to increase.

The architecture of applications software systems is increasingly important in how systems support user needs. The majority of systems that have been created outside of the telecommunications industry are integrated using procedural and other paradigms that often lead to ineffective solutions. In fact, for systems created by corporate development organizations, a majority of the software projects are considered unsuccessful at completion. From an architectural perspective, many of these systems resemble the configuration in Figure 3.4 (a) for stovepipe systems. In a stovepipe system, there are a number of integrated software modules. Each software module has a unique software interface. This unique software interface corresponds to a single program implementation.

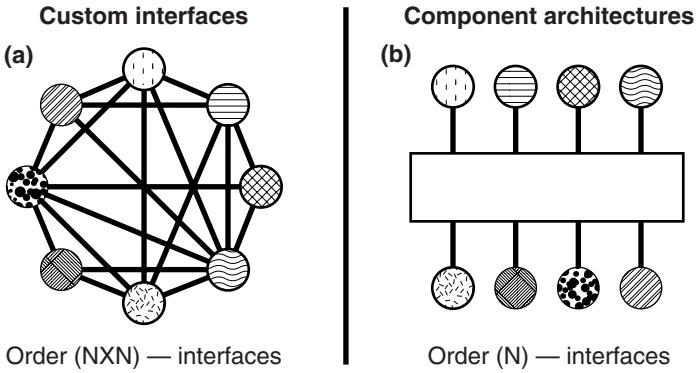

FIGURE 3.4 (a) Stovepipe Systems and (b) Component Architectures

When the system is integrated, there are many one-to-one dependencies between various parts of the system. These dependencies are unique integration solutions. As the scale of the system increases with the number of modules, the number of dependencies increases by the square of the number of modules. This increase in complexity has many negative consequences. In particular, as a system evolves, it becomes increasingly less amiable to modification and extension. System extension happens to be one of the major cost drivers in application development; it can account for as much as half of all software cost [Horowitz 1993].

An alternative way of building systems includes a planned definition of software interfaces that provide a greater level of uniformity across the integrated solution. Component architectures are application systems that are defined using consistent application program interfaces across multiple instances of software subsystems (Figure 3.4). Component architectures reduce the dependency between software modules. The reduced dependency enables the system to be extended and support larger scales of integration. The complexity of the software integration of a properly built component system is tied to the number of software modules needed to build the system.

3.4 CLIENT-SERVER TECHNOLOGY

Client-server technologies are the result of the evolution of software technology supporting application systems. In particular, the evolution of client-server technologies has been an important factor in the expansion of information technology across an increasing range of application business processes. Originally

59 technologies focused on file sharing. File sharing is still the dominant paradigm of the Internet today with protocols such as HTTP supporting access to the global file systems available. File server technologies evolve into a second generation of capabilities dominated by a database server technology. It is important to note that the file server technologies have been closely linked with the evolution of distributed computing technologies.

Increasingly, client-server technologies are being replaced by *N*-Tier component-oriented solutions. Based upon Java application servers, the *N*-Tier solutions include support for thin-client user interfaces with increased scalability and reliability.

One of the most successful networking technologies came from Sun Microsystems and was called network file server. Sun Microsystems was successful in encouraging the de facto standardization of that technology by providing free reference technology access in terms of source code for implementation on arbitrary platforms. Network file server technology is based upon open network computing, another Sun Microsystems technology that was one of the first successful generations of distributed computer technology. The network file server was a procedurally based technology closely tied to the C programming language, as was the other important remote-procedure-call technology called the distributed computing environment. Both technologies resulted in file-sharing capabilities that were widely implemented. The database server technologies utilized these underlying distributed computing capabilities to provide remote access to database systems from a variety of client platforms.

Another important technology that arose during the database generation was that of transaction-processing monitors. Transaction-processing monitors enable the consistent and reliable maintenance of data integrity across distributed systems. Transaction-processing technology continues to be an important add-on capability to distributed computing technologies to ensure robustness and integrity of implementations.

Groupware technologies also arose in the late 1980s and early 1990s starting with email and evolving to higher forms of interactivity, some of which can be seen on the Internet today (e.g., chat rooms and videoconferencing). Recently, the technologies of object orientation, distributed computing, and the Internet have merged to support adaptable computing environments that can scale to global proportions. This generation of technologies is supported by application servers primarily based on Sun's Java or Microsoft's .Net platform.

The client-server technologies initially arose as an evolution of mainframe-based technologies. Mainframe-based technologies were a natural out-

growth of single-processor systems that date back to the origins of computing. In a mainframe technology, the processing and management of data in the system is completely centralized. The mainframe is surrounded by a number of peripheral client terminals that simply support presentation of information. In the client-server generation of technologies, the client computer has become a significant processing resource in its own right. Client systems that arose during the personal computer revolution are now capable of processing speeds that rival and greatly exceed that of former minicomputer and mainframe computer generations. Initially, in order to support access to data in departments and enterprises, client-server technology supported the connection through local area networking to the back-end mainframe minicomputer and workstation server systems. The technology at the software level supporting this communication is called *middleware.*

> *"Genius might be that ability to say a profound thing in a simple way."*— Charles Bukowski

History

Initially, middleware was installed as a custom capability to support client-server networking between PCs and server platforms. As technology evolves, middleware is becoming embedded in the operating system so that it is a natural capability of client platforms as well as server platforms. Client systems with embedded middleware can now support on-board services to applications running locally and across the network. This evolution of client-server technology to an embedded capability has added many new challenges to the implementation of application systems. In fact, there are various antitheses to the client-server evolution, including a resurgence of the mainframe platform as a significant business of IBM and the capability called the *network computer* which begins to resemble the dumb terminal of mainframe days (Figure 3.5).

Object technologies are organized around client-server capabilities. Object technologies come in two primary categories. Some are organized to serve the process of software development. Examples of these technologies include object-oriented analysis and object-oriented design. Object-oriented analysis comprises the definition of information technology capabilities that are models of current and future business processes. Object-oriented modeling provides rich capabilities for representing business entities and business processes. This is in contrast to procedural and relational database technologies, which require the application designer to compromise the representation of the business environment to the constraints of the technology in terms of control flow and data

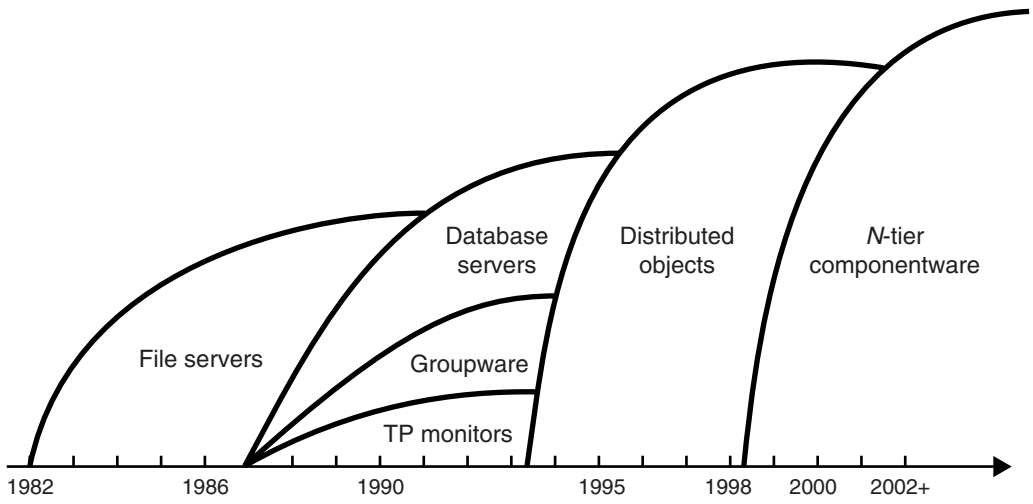

FIGURE 3.5 Origins of Client Server Technologies

representation. Object-oriented analysis, because of the natural correspondence of state information in process, provides a mechanism for modeling reality that is relatively easy to communicate with end-users. Because end-user communication is facilitated, the design and validation of object-oriented systems is greatly enabled.

Object-oriented design is another major software phase that has been successful commercially in the software process market. Object-oriented design comprises the planning of software structure and capabilities that support the reduction in software defects and rapid prototyping of software capabilities.

The other major category of object technology focuses on implementation. At the center is object-oriented middleware technology. Object-oriented middleware supports distributed computing and the integration of various heterogeneous software technologies including operating systems, programming languages, and databases. Object-oriented programming languages are the direct expression of the object paradigm. Object-oriented programming languages support the encapsulation of data with process in the form of abstract data types in component objects. There are as many object-oriented programming languages as there are procedural languages. The predominant languages for object-oriented programming include C++, the Java language, and increasingly C#, but a significant number of communities support Eiffel and other languages. Object-oriented middleware allows these languages to interoperate to form applications. Object-oriented programming languages are one possible choice for implementation of application software. It is also possible to utilize

object-oriented analysis and design to support programming in procedural languages. This occurs frequently, as many corporate development environments use procedural languages for their mainstream languages, such as the C programming language and COBOL.

One of the important qualities of object orientation is that the developer should not have to be concerned about the underlying implementation. If the underlying implementation is procedural or is object-oriented, it should not and does not matter if the applications are properly encapsulated. Distributed object middleware supports the opaque encapsulation property, which makes this possible. The integration of commercial software with legacy and object-oriented applications is also enabled as a result of these encapsulation properties (Figure 3.6).

Object-oriented middleware technologies can be viewed as an outgrowth of their procedural producers. Beginning with operating systems, procedural technologies supporting interprocess communication were added to enable file sharing and the evolution of client-server capabilities (Figure 3.7). Some of these technologies include the Remote Procedure Call (RPC) technologies such as Open Network Computing (ONC) and Distributed Computing Environment (DCE). The RPC technologies were preceded by socket-level technologies,

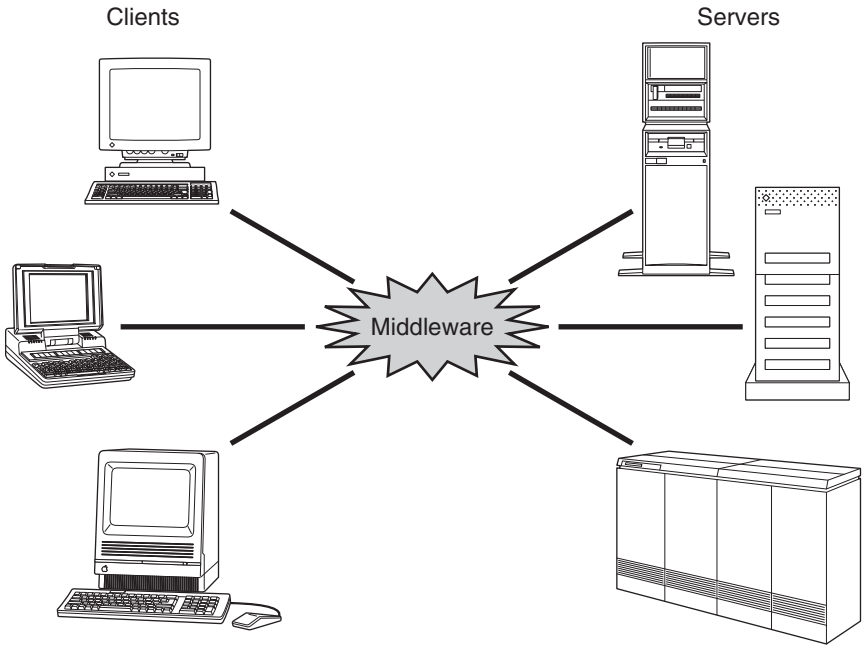

FIGURE 3.6 Role of Middleware

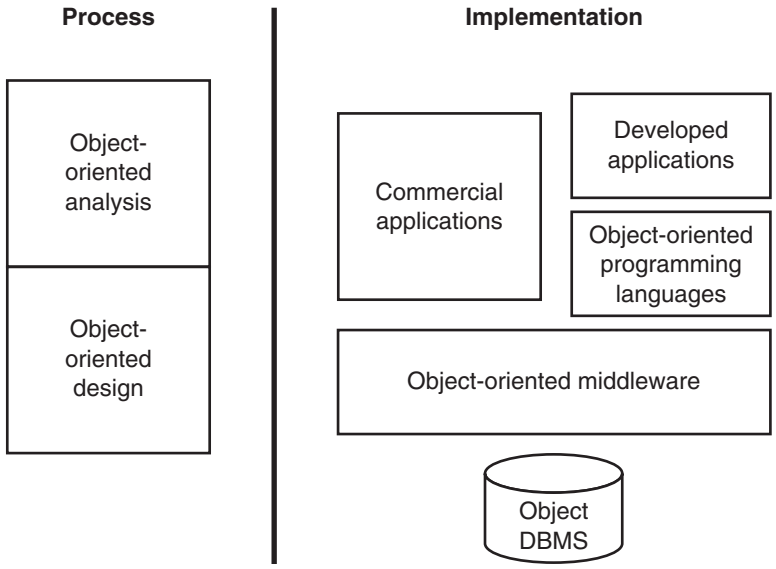

FIGURE 3.7 Middleware Reference Model

which are a more primitive form of messaging. Today, all these technologies are still used actively in application systems and on the Internet. The object-oriented middleware technologies provided a next generation of capabilities, which bundled more of the application functionality into the infrastructure.

Distributed Components

It is interesting to note that previous generations of interprocess communication technology were marketed with the promise of universal application interoperability. Component-oriented technology is marketed the same way today. Distributed object-oriented middleware has the advantage of retrospection on the shortcomings of these previous technology generations. It was found that even though remote-procedure-call technologies enabled the integration of distributed software, the primitive level of these technologies required substantial application programming in order to realize systems. Once the systems were implemented, the systems tended to be fairly brittle and difficult to maintain.

Microsoft, in 1996, released DCOM as a multimedia middleware technology for the Internet. DCOM still exposed many of the lower level primitive details, which were the downfall of remote procedure calls. DCOM added some object-oriented capabilities and a natural integration support for C++ programming. Simply adding the capability to support C++ didn't necessarily

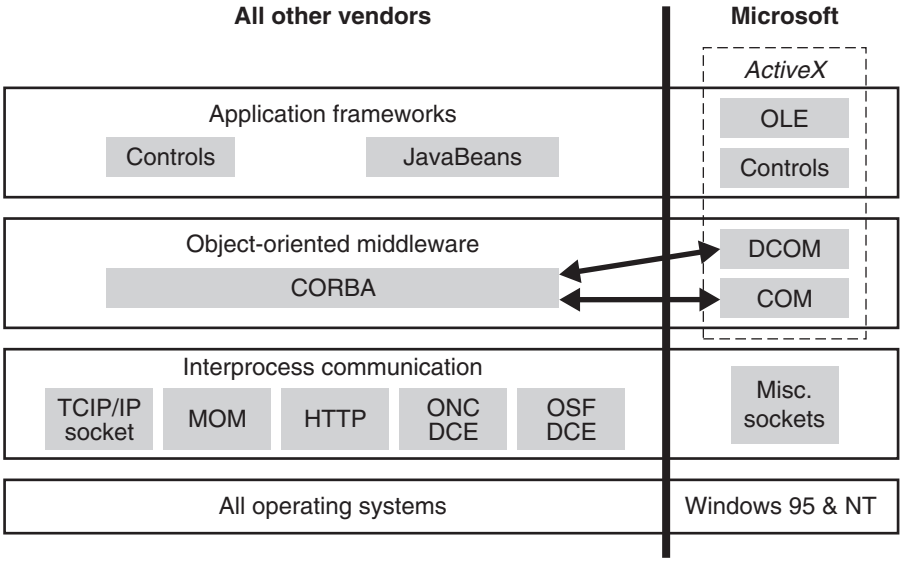

FIGURE 3.8 Distributed Technologies in Context

overcome the procedural route that exposed excessive complexity to distributed system developers in the DCOM predecessor called the distributed computing environment. However, with the release of the current generation of product, Microsoft .Net, Microsoft has created an enterprise development environment that can compete favorably with the capabilities of the more mature J2EE application servers.

The Common Object Request Broker Architecture was the first technology to be designed from the ground up to support distributed object-oriented computing. Figure 3.8 shows that there is a partitioning of a technology market between the Microsoft technology base and virtually all other information technology vendors. The other vendors support various open system technologies that are the result of consensus standards processes. CORBA is universally accepted as the vendor-independent standard for distributed object middleware. CORBA simplifies distributed computing in several ways. The most significant advance is the language independence that CORBA provides, allowing multiple programming languages in heterogeneous environments to interoperate using object messaging.

Above the middleware layer are other technologies that support further integration of application functionality. In the Microsoft technology base, these technologies have been grouped into a brand name called .Net. The .Net technologies include a substantial reinvention of middleware capabilities that elim-

inate interface definition languages. The CORBA capabilities are widely available today and support multiple programming-language integration from multiple vendor platforms.

CORBA technologies are the product of an open systems consortium process called the object management group, or OMG. The OMG has over 700 member organizations including all major vendors in information technology, such as Sun Microsystems, Hewlett Packard, IBM, Netscape, and Microsoft. The OMG has addressed the problem of application software interoperability by focusing on the standardization of programming interfaces. With previous generations of remote-procedure-call technologies, the only widely adopted standard interface was the network file server, which is really the most primitive form of software interoperability beyond exchange of removable media. It is important for end-users to provide their requirements and interact with open systems processes because they shape the form of technologies that will be used for end-user system development. In particular, sophisticated users of technologies can encourage open systems consortia and software vendors to provide more complete capabilities to enable the development of complex systems. This reduces technology risk and creates more leverage for application developers.

The CORBA technologies are centered around the object request broker that the component standardizes (Figure 3.9). In the object management architecture, which is the route node of OMG diagrams, there are several categories of objects. The object request broker is distinguished because it is the object through which all the other categories of object communicate. The object management architecture is conceptually a layered architecture that includes increasing levels of specificity for domain application implementation. The most common capabilities embodied by object technologies are standardized through the object request broker. The next level of capabilities is called the CORBA services, which provide enabling functions for systems implementation. The CORBA services are comparable in functionality to current operating-system services that are commonly bundled with platforms. The CORBA services provide the first step toward a distributed operating system capability that supports the integration of application software and commercial software across all types of platforms and environments.

CORBA technology is widely available today and is a mainstream technology available on virtually every operating-system platform. Some of the more innovative platforms, including the Netscape Communicator, which could be considered an operating-system platform in its own right, are bundling CORBA with all of their deliverable licenses. Microsoft also supports the CORBA technology market by delivering specifications that enable inter-

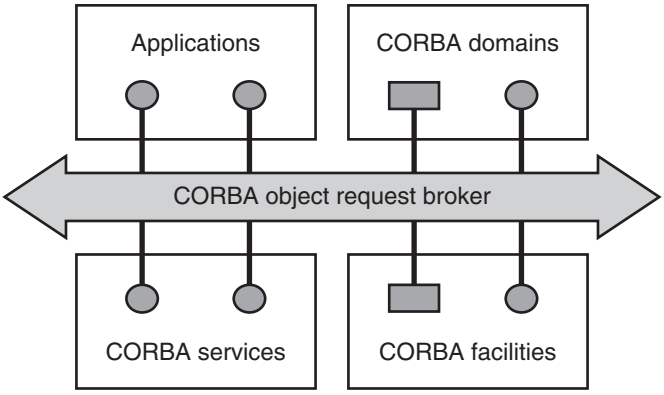

FIGURE 3.9 Object Management Architecture

working with the Microsoft infrastructure workings. The OMG has standardized interworking specifications for both COM and COM+ generations of Microsoft technologies. These standards are available on products on major CORBA implementation systems today.

In addition, third-party vendors are providing direct support for CORBA. These include vendors like Black and White software, which provide graphical user interface development tool kits, database vendors, system management vendors, and specialty market vendors such as real-time and computer-aided software engineering tools. CORBA provides the interface definition language, which is the key capability fundamental to object orientation. The interface definition language is a notation for defining software boundaries. IDL is a specification language that enables the description of computational software architectures for application systems as well as international standards for interoperability.

The interface definition language from CORBA has also been adopted by the international standards organization and the formal standards counterparts for telecommunication systems. IDL is the international standard DIS14750. As such, IDL is a universal notation for defining application program interfaces in software architectures. Because IDL is programming-language independent, a single specification will suffice for defining software interfaces on any language or platform environment. IDL interfaces support object-oriented designs as well as the integration of legacy software. Since the object management group is the only major standards consortium developing object-oriented standards specifications for software interfaces, IDL is synonymous with object technology open system.

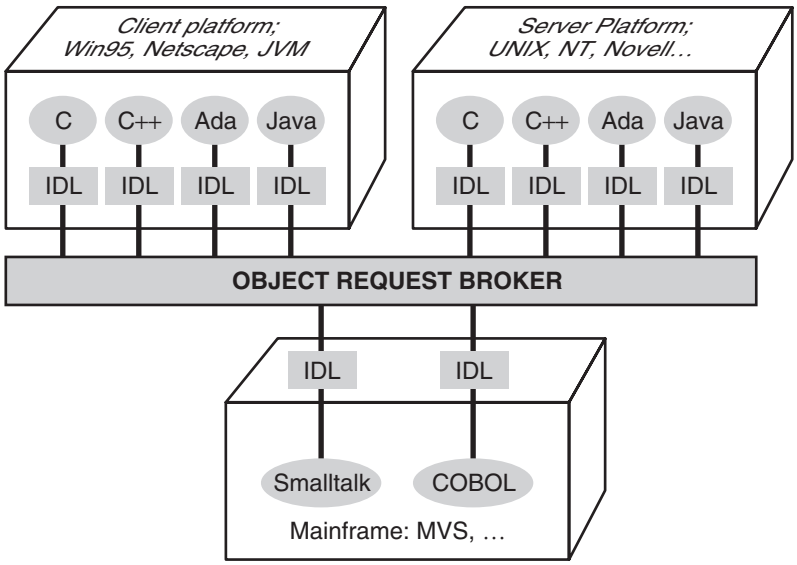

FIGURE 3.10 Technology Independence of the Interface Definition Language

IDL supports the integration of a diverse array of programming languages and computing platforms (Figure 3.10). With IDL, one can specify software interfaces that are compiled and readily integrated to available programming languages. These capabilities are available commercially and support distributed communication in a general manner.

This section discussed how mainframe technology has evolved into client-server technologies with middleware providing the distributed computing software capabilities. Because client-server technologies have merged with object technologies, it is now possible to provide object-oriented capabilities that augment legacy systems across most or all programming environments. In addition, interoperability between CORBA and the Microsoft counterpart called COM+ enables the coverage of popular platforms on many organizational desktops. The vendors supporting open systems also support CORBA. The dominant Internet vendors are delivering CORBA and associated protocol stacks to numerous licensees. CORBA is the standard for object-oriented middleware. The products are available now as are the horizontal services specifications that enable application development. The OMG is proceeding to develop the vertical specifications that will provide direct support for application-level interoperability.

The ISO has supported the designation of CORBA IDL as a standard for the definition of software interfaces across all computing environments.

Object orientation is a natural paradigm for modeling real-world information and business processes. Object technology supports the integration of heterogeneous and distributed information technologies that include legacy systems (Figure 3.11). Combining object orientation and component technology enables the creation of ambitious system concepts, which are increasingly becoming the competitive advantage of application companies and end-users.

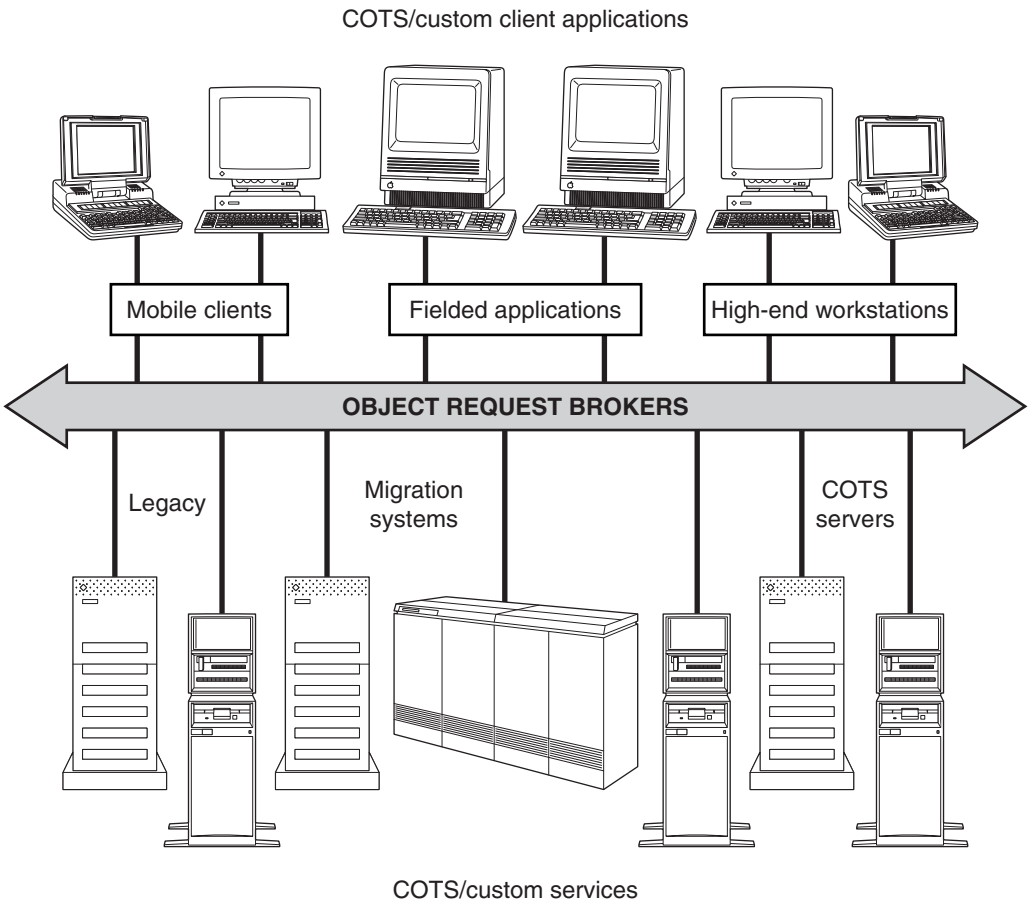

FIGURE 3.11 Interoperability Vision for Object Technology

3.5 INTERNET TECHNOLOGY

eXtensible Markup Language (XML)

Few technologies have generated as much confusion in the mainstream press as eXtensible Markup Language (XML). While XML is a fundamental, enabling technology, there is a tendency to combine it with other technological solutions and blur XML's capabilities with other, often proprietary solutions. Here the heart of what XML is and why it is likely to have an exceptionally long technology life cycle will be discussed.

XML was created to put an end to application-specific data interchange problems. Rather than allow two or more applications to decide among themselves what format to use for data interchange and code the logic for reading and writing to the agreed-upon format in each and every application, XML provided the means to describe data in an application-independent manner. The XML file uses tags to include not just the application data but information (tags) to describe the data as well. XML is a World Wide Web Consortium standard and is used by a huge number of other standards.

One of the benefits of XML is that it addresses the fact that prior to its development, coding import and export routines for every application that interacted with a data set was expensive and brittle. Every time the data changed, every application that interacted with the data had to be modified to understand the new data format even if the change had little impact on the data elements used by the various applications. Applications could not extend existing data formats without ensuring that all applications were also upgraded to handle the changes. Essentially, data were modeled in lockstep with the applications responsible for reading and writing the data. This added substantially to the maintenance costs of all applications that shared data, which tended to include most applications in many environments. XML was revolutionary in that it provided a simple means to model data independently of the application(s) that used the data.

XML is ultimately a data format. The original XML 1.0 specification was quite concise and mostly defined a way to use tags to describe data elements. The tags are user-defined and intended to have the following characteristics:

Structured. XML uses tags to describe the data making the data file self-describing. Programs that read and process XML documents can easily determine if a document contains particular data elements. Also, it is easy for a program to determine whether an XML document is cut off or poorly formed.

Flexible. For any collection of data, XML provides several ways to represent the data. As always, flexibility is both good, in that it allows developers to make appropriate choices for how to represent data within an XML file, and bad, in that it also allows developers to make inappropriate and unwise choices regarding data representation.

Validated. A Document Type Description (DTD) or XML Schema lets developers define rules that guide the representation of data. XML parsers are widely available to validate document correctness against the schema.

Adaptable. The applications that produce the XML files, the operating systems, programming languages, and data management systems can all change, and the XML files will still be readable.

Standard. The use of XML requires no license, and no corporation can change it to make it incompatible with other applications.

Readable. An XML file can be edited, modified, and stored as plain text.

As an example, creating an XML document to describe flavors of ice cream is as easy as deciding what is to be described and then documenting the specific instances.

Why is this technology so powerful? Unlike other data formats, even this simple XML document will be understandable twenty, fifty, and maybe hundreds or thousands of years from now. Few data formats in use even ten years ago are understandable by today's applications. And if data are understood, then they can be used/processed. Additionally, with XML parsers and other complementary technologies, much of the conversion process between different XML formats (and other formats) can be automated.

However, there is a tradeoff for this flexibility. XML is a very verbose way of describing data. Few data formats have the space requirements of an XML document when storing or transferring the identical information content. As a result, other data formats are desirable when performance or storage space is a constraint. Of course, with the current pace of advances in hardware processing and transmission speeds, the size of XML files is frequently only a very minor consideration. A larger problem arises in managing large number of XML documents. Searching large number of XML documents is often problematic. However, XML document indexing systems and even XML-specific hardware have helped mitigate issues with searching large numbers of XML documents. Some database vendors are also implementing XML types in their databases to handle storage and searching issues; for example, such enhancements are available for Oracle 9i in their XML database products. A number of other companies have developed XML-specific databases with a number of

customizations designed to increase performance in searching and efficiently storing XML content. Finally, be aware that while many of the claimed benefits with XML are indeed possible, they are very seldom automatic. The use of XML itself does not alone accomplish a great deal; rather, considerable thought, planning, and design are required to use XML effectively in conjunction with other technologies.

Sun Microsystems J2EE and Microsoft's .Net

First, the basics about the two major enterprise development platforms in the industry: Sun Microsystems Java 2 Enterprise Edition (J2EE) and Microsoft .Net. J2EE are Java-centric operating environments that strive to be platform neutral. This means that J2EE developers all work with the Java programming language; however, the language itself is portable across all the major hardware environments and software operating systems. In contrast, Microsoft's .Net environment supports multiple programming languages but is currently focused on a development environment that executes on the Microsoft Windows line of operating systems. In addition, J2EE is primarily a set of specifications that are implemented by a number of different vendors. Microsoft .Net is sold as a suite of products based on proprietary Microsoft intellectual property. This basic difference in strategy has resulted in an interesting polarization in most of the software industry. Sun, along with a few major players such as IBM and Oracle, has made a tremendous investment and gathered a majority of vendors in support of the J2EE standards. Most of them have integrated J2EE in their flagship product lines, and many vendors have their own implementation of J2EE. In the other camp, Microsoft with its .Net has leveraged Microsoft's existing vendor relationships to create a rapidly growing sphere of vendors supporting the .Net line of solutions for enterprise software development. They willingly conceded the fundamental .Net set of core services to Microsoft and modeled businesses toward profiting from value-added software based on the core infrastructure.

The J2EE environment requires all components to be written in the Java programming language. The Java Virtual Machine (JVM) compiles programs written in Java into Java-specific byte code, and the JVM executes the compiled byte code at runtime. This contrasts sharply with the .Net approach, which uses its Common Language Runtime (CLR) engine to compile a number of programming languages into a truly language neutral intermediate code. The CLR enables developers to use any of the programming languages supported by the .Net development tools and has defined mechanisms to enable .Net components written in one language to be called from any other supported programming language easily. This allows multiprogramming language devel-

opment in a manner that had not gained mainstream acceptance before its introduction in .Net. The jury remains out on precisely how desirable this capability will be within the industry, but certainly the appeal of reusing a tremendous body of existing code regardless of its implementation language has an interesting surface appeal.

In terms of maturity, the J2EE environment has a much longer history and greater industry involvement in its development and evolution. J2EE has been successfully deployed in numerous mission-critical applications in some most challenging vertical industries including financial services and telecommunications. Plus, since J2EE solutions are available from many different vendors, there is a greater variety in the available implementations, toolsets, and value-added products. Unfortunately, as a result of dealing with a number of vendor solutions, care must be taken to deal with product interoperability issues, which are less of a problem than with single vendor solutions like .Net.

Overall, J2EE offers an experienced development team many advantages because J2EE provides greater programmer flexibility than .Net and can enable the development of well-performing, highly customized applications. However, this greater flexibility also allows the easier introduction of serious errors in areas such as memory and resource management. It is far easier to get a multi-tier application up and running in .Net initially, but J2EE enables the experienced developer greater freedom to develop more powerful applications. The .Net architecture is more focused on ease of use and thus doesn't provide as much access to lower level state management and caching techniques frequently utilized in J2EE application development.

Web Services

XML is a data format that enables the data exchange between heterogeneous computing environments. Web Services are important because they use the Internet as the data transmission layer to enable the sharing of distributed processes. By using the Internet, these processes are widely available to potentially millions of users and can serve as the building blocks for larger distributed applications. A Web service can be implemented in any language on any computing platform. The service interface is then described in XML using the standard Web Services Description Language (WSDL), a specialized XML schema for Web services. A client to a Web service can invoke WSDL using the standard Simple Object Access Protocol (SOAP) that provides an XML format for representing the interface of a Web service and invoking it over HTTP. Since HTTP is the standard protocol for the Internet, Web services can be deployed and used over the existing Internet infrastructure. The Web service

provider can also register their Web services in a Universal Description, Discovery, and Integration (UDDI) registry to enable clients to search, discover, and access their Web service dynamically.

The basic technologies to enable the creation and use of Web services are widely agreed upon industry standards supported by both the Microsoft .Net and Sun J2EE platforms. However, the value-added services relating to security and electronic commerce are still in progress, and the potential of Web services will not be realized until these key technologies are agreed upon by and widely adopted throughout the industry.

3.6 ARCHITECTURAL LAYERS AND WHEN TO USE THEM

A two-tier system (Figure 3.12) is one where user interface and application code is allowed to access the APIs directly for database and network access. The application uses the data model stored in the database but does not create a logical model on top of it. A two-tier application is ideal where the system under development is a prototype or is known to have a short system life cycle where APIs will not change. Typically, this approach is used for small applications where development cost and time are both small.

In addition, a two-tier system makes sense in a component-oriented development environment where this approach is used in the implementation of a particular component. The component interface provides a layer of insulation against many of the negative consequences of this approach. Many applications that are created using scripting languages fall in this category as the development of multiple architectural layers could be cumbersome to the point of being impractical.

Finally, a two-tier approach will provide better performance and less need to add mechanisms that control resource locking. While it will have less of an upside in its ability to scale to many concurrent users for limited use applications, the simplicity of a two-tier model may far outweigh the benefits of the other alternatives. In addition, frequently database-stored procedures may be used to eliminate some of the simpler shared data issues by adding common data processing routines into the database application.

Three-tier applications have been common since the growth of the database. A three-tier system (Figure 3.13) satisfies the need for implementation isolation. Most frequently, this is desirable in any system where the

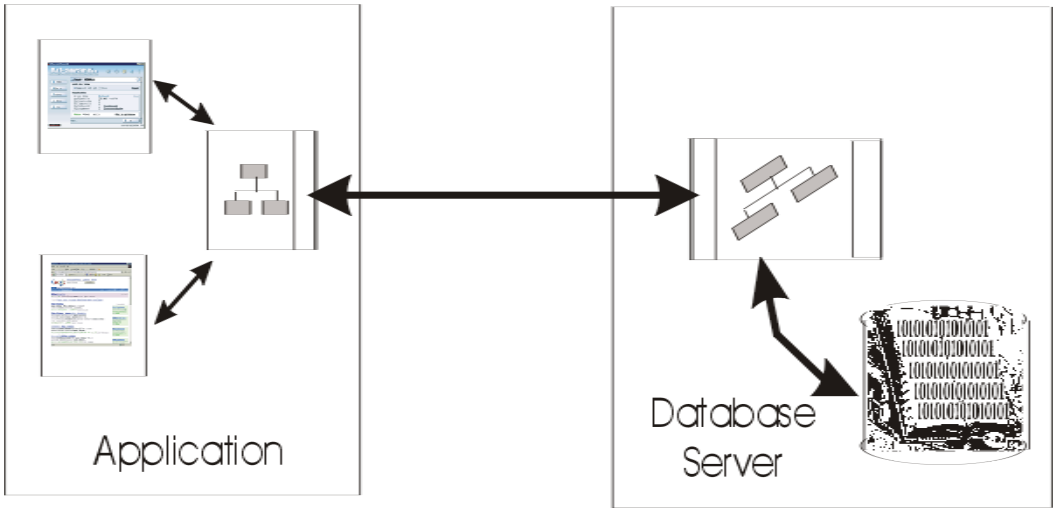

FIGURE 3.12 Two-Tier Layered Architecture

storage/database layer of an application may need to be changed. However, this technological isolation is not restricted to just databases. It can, and should, be used whenever it is valuable to share code without requiring the application developer, or more importantly, the application maintainer, to have a detailed understanding of the implementation details of the lowest layer.

Quite frequently reuse is a major design consideration where the application model is created to allow part of it to be reused by multiple user interface view components. As a guideline, whenever an application needs multiple views of the same data, a developer should consider utilizing a three-tier approach instead of a two-tier approach.

Major issues to consider in moving from a two-tier model to a three-tier model include the availability of appropriate network resources and a locking solution to manage concurrent access to data.

A more recent trend has emerged as a result of an increased emphasis on network computing, and that is the four-tier system (Figure 3.14). A four-tier system is an alternative to consider when the application layer needs to support advanced behavior. A four-tier model is like a three-tier model where the application layer is split into a presentation layer and a session a layer. The presentation layer assumes the view portion of the application model along with the application logic that is constrained to the operations of a particular view. The session layer handles resources that are shared between presentation components, including communication with the potentially distributed business object model.

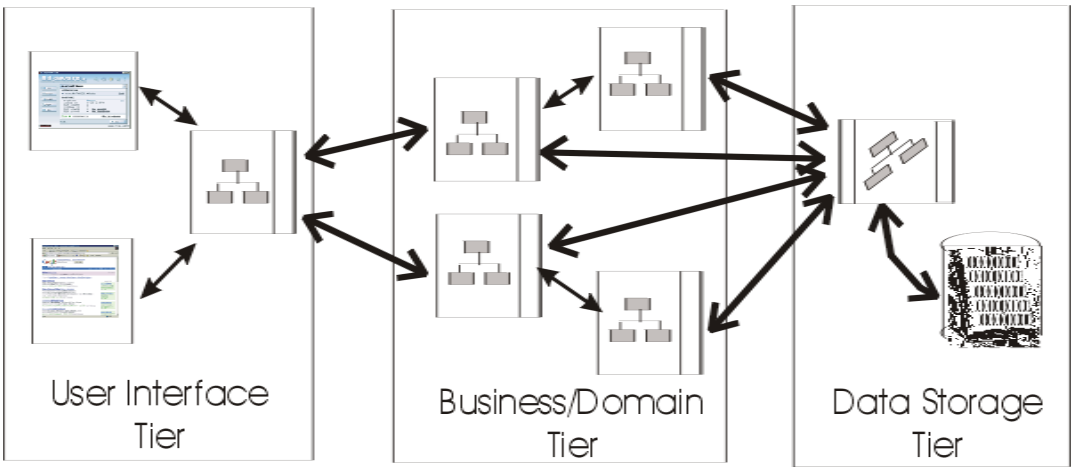

FIGURE 3.13 Three-Tier Layered Architecture

A four-tier development approach is needed when there is a significant amount of coordination required between presentation components as well as a requirement that many resources be shared between them. For example, it works well when caching is required for performance reasons. A session layer allows many different presentation components to take advantage of the performance gains caching provides. Also, if a client is forced to make multiple, potentially complex distribution decisions, it makes sense to encapsulate that logic in a session layer of the application.

Factors that may indicate the need to consider a four-tier development approach are many. Obviously, any four-tier system should be large with a long expected life cycle. Reuse of existing components and subsystems is frequently a sufficient reason to incur the overhead associated with a four-tier system. Along the same lines, environments where individual components are expected to change frequently the design goal is to insulate the majority of the system from changes in component implementations. A four-tier approach provides support for incremental migration of components and subsystems across technologies, both legacy and new. Also, a four-tier system can be more scalable than a three-tier system.

Other factors to consider include systems where the reliability of components is either unknown or variable. A four-tier system can easily incorporate runtime discovery mechanisms to roll over to different component implementations in the event of intermittent component failures. Many complex systems with four or more tiers provide at least some capability to discover new capa-

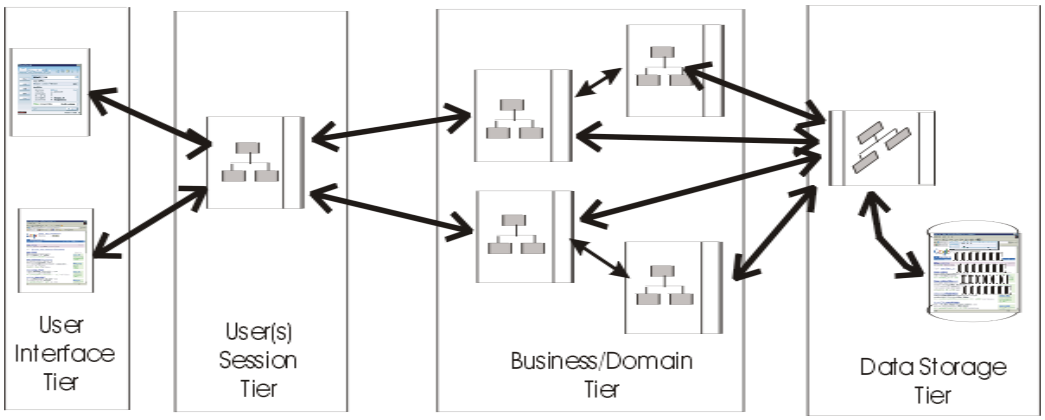

FIGURE 3.14 Four-Tier Layered Architecture

bilities (e.g., they implement a UDDI registry for advertising new Web Service implementations). If the environment utilizes multiple, potentially conflicting technologies, a four-tier system provides mechanisms to manage differences in either the session management layer or in the business domain object layer. Also, a four-tier model may be desirable if the client has several diverse application models that all need to share common data resources. Frequently, some application components will be content to allow the business domain components to handle resource management issues and can afford to wait for most resources where others may not want to block and wait for resources and have to manage client access in the session layer.

A peer-to-peer (P2P) architectural approach is ideal for systems that need to be highly scalable. Also, they are useful when distributed components need to cooperate to accomplish a task and the reliability of communications and other components are variable. It is important when developing P2P systems that the operating environment be well understood because sloppy practices could result in major disasters. Also, when utilizing P2P technologies, it is important that the interfaces be standardized and highly unlikely to change. Having to cope with multiple incompatible versions of a P2P network is a nightmare.

N-Tier and/or a combination of these approaches (Figure 3.15) should be used only for significantly complex systems made up of subsystems and components that have differing software life cycles. This is true of most large-scale heterogeneous enterprise systems where, at any given time, components are being upgraded, replaced, or added to the system. With such a system, consideration must be given to the administration of the system components.

What are the features that may merit the complexity of an *N*-Tier system? In general, it includes systems that manage a variety of data to enhance the user experience. The requirements would include Web sites and applications that remember the profile information of users, allow users to set preferences that control Web pages and applications, manage complex security requirements such as access control lists for controlling resources, and allow users to make changes that require storage management and rule execution within the back-end applications.

With an *N*-Tier application, the application functionality is partitioned into a number of logical layers that can be maintained and deployed separately. The functionality of each layer is less standard than that of three-tier applications, and frequently many layers can be grouped together to provide presentation, application, and/or business logic and storage management functionality. The primary benefit of supporting many layers is that it is easier to make changes in one layer without having to alter many or, preferably in most cases, any of the other layers. Additionally, the application can be scaled to handle great loads of users and/or data by altering the distribution or load management of one or more layers. Frequently, this scaling can be transparent to other layers and even automated in many cases. In fact, often multitier is assumed to mean spreading processes across various processors and machines rather than defining software boundaries within an application.

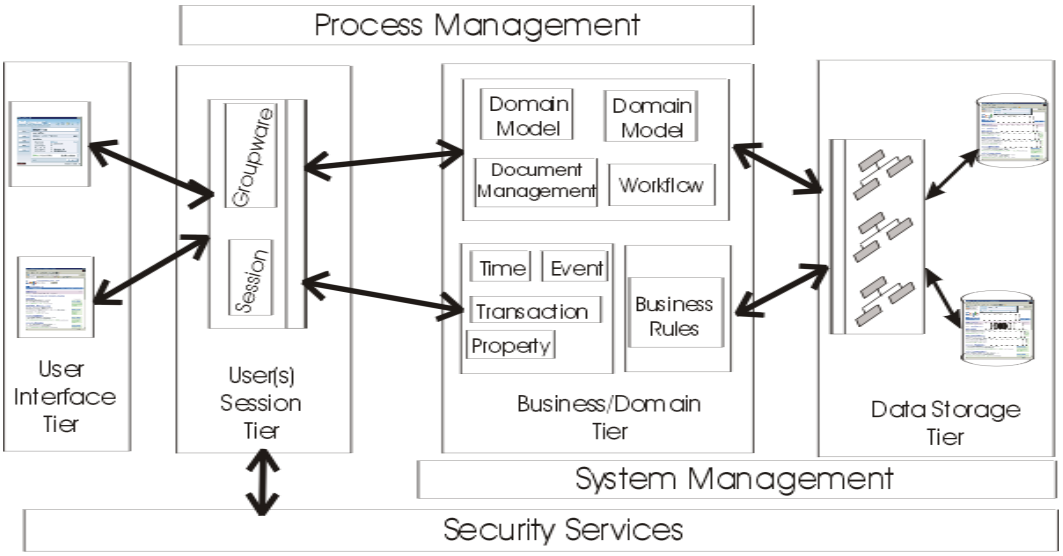

FIGURE 3.15 Combination of *N*-Tier and Peer-to-Peer Architecture

3.7 SOFTWARE APPLICATION EXPERIENCE

In the commercial end-user environment, object technology has been applied to many important applications that enable business advantages. Examples include Fidelity Investments, one of the world's largest mutual fund companies, which as many as five years ago integrated its fund management workstations to support the integration of multisource information including decision-support capabilities that are crucial to the fund management business. The infrastructure they chose was an object request broker implementation conforming to the CORBA standard. Using CORBA, Fidelity Investments is able to customize the information gathering and analysis environment to the needs of individual fund managers. Many readers of this book probably have funds invested in one or more of the securities supported by CORBA. Wells Fargo, a large banking institution, has also applied object technologies to multiple applications to derive competitive advantages. One example is a financial transaction system that was developed, prototyped, and deployed in less than five months based upon an object technology and CORBA implementation. In that system, they integrated mainframe environments running IBM operating systems with minicomputer environments serving the online transaction terminals. In another Wells Fargo application, they integrated heterogeneous systems to support system management across a large enterprise. System management is one of the challenging and necessary applications that the client server has created because the operation and management of information technology is no longer centralized but still needs to be coordinated across many autonomous departmental systems as well as user desktops. Wells Fargo took advantage of object technology to implement such a distributed system management capability, and greatly reduced their expense and response capabilities for system support challenges.

Another dramatic example of object technology was implemented by a large insurance provider. USAA had an auto claims system that customer service agents used to receive reports of damage claims over the telephone. USAA, in addition to auto insurance, has a number of other related product lines including life insurance and loan capabilities. By integrating their information technology using objects, USAA was able to provide the customer service agents with information about the full range of USAA product lines. When a customer called with an auto damage claim and the car was totaled and needed to be replaced, the customer services agents were able to process the insurance claim and offer a new car loan for the replacement of the vehicle. In addition, the customer service agent had information about customers such as the ages and number of children and was able to offer additional insurance coverage at

the appropriate time frames during this same auto claim call. With these enhanced capabilities, essentially reengineering its customer service process, USAA was able to realize 30% increased revenue on its existing customer base by providing additional services to the customers who were calling USAA for auto claims purposes.

In the public sector, object technology has also been applied and has delivered significant benefits. Several examples were implemented through the work of the authors on the Data Interchange and Synergistic Collateral Usage Study (DISCUS) project. This project and its lessons learned are described in *The Essential CORBA* [Mowbray 1995]. One of the first lessons learned on discus was the power of using object technology to reuse design information. Once software interfaces were established and specified using IDL, it was relatively inexpensive to have contractors and commercial vendors support interoperability interfaces. The discus capabilities were defined before the Internet revolution, and when it became appropriate to integrate Internet capabilities, the same encapsulations were equally applicable to integrating new ways of viewing the data through Internet browsers. The existing legacy integrations implemented by discus were then used to extract information for viewing on Internet browsers.

Another case study implemented by the authors involved a set of information access services, which is a case study documented in *Inside CORBA* [Mowbray 1997c]. In this application, the fact that the government had implemented a variety of systems with similar capabilities and the end-users needed these systems to interoperate and support expanded access to information resources is examined. The application described in this book does not differ in substance from the environment required by the Fidelity Investment Managers—in other words, gathering information from diverse resources in order to support important decisions. To resolve the users' needs, the authors conducted a study of existing systems that focused on the software interfaces supported through multiple technologies. By learning the details of the legacy system interfaces, new object-oriented designs could be formulated wherein the existing functionality was captured in a manner common across the legacy system environment. By committing the new interface design to an IDL specification, other contractors could be used to help implement prototypes and forward the specifications through government standardization processes. Within two years, the interoperability concept evolved from ground zero to working software including a formal test sweep that assured conformance between multiple implementations of the specification.

Many enterprises have the opportunity to realize these kinds of results. Because information technology in large enterprises is evolving from desktop

and departmental information systems to interoperable enterprise systems, a layer of enterprise architecture that does not exist in most organizations can be implemented using distributed-object technologies in a manner that provides interoperability in a general way.

In summary, commercial organizations have realized from object technology many benefits that are directly relevant to their corporate competitive advantages. The authors' experiences in research and development show that design reuse is one of the most important concepts to apply in realizing these kinds of results. Given a proper software interface specification, software developers can relatively easily understand the specification through training processes and then proceed to implement the specification. A much more difficult problem would be to ask developers to integrate systems without this kind of guidance. In other words, reinventing a new custom interoperability link is significantly more difficult than if the developers are given a design for how the systems interoperate and simply need to implement the code to implement that capability. In the course of research and development, the authors discovered these kinds of benefits even at the smallest scales where only two or three subsystems were being integrated; as the scale of integration increased up to seven or ten or more systems, the benefits also increased.

Systems interoperability is achievable today through object technology, and these benefits are being realized in existing commercial systems and in system procurements in the public sector.

3.8 TECHNOLOGY AND APPLICATION ARCHITECTURE

Software architecture involves the management of both application functionality and commercial technology change. The kinds of benefits that were just mentioned are not the direct result of adoption of a particular technology but involve exploiting the technology in ways that are most effective to realize the business purpose of the system. The simple decision to adopt CORBA or COM+ is not sufficient to guarantee positive business outcomes. One of the key challenges is managing the change in commercial technologies in a manner that supports long-term system life cycles and extends the system without substantial maintenance as the commercial technology evolves.

Figure 3.16 is an example of the class of technology challenges that must be managed by object-oriented architects. Figure 3.16 concerns the evolution of middleware technologies, starting with the socket technologies and evolving

into remote procedure calls and distributed computing environment to the current J2EE and ActiveX technologies. No one can reliably predict the future, but given what is known about proprietary technology evolution as well as open systems evolution, it is likely that many of the technologies that are becoming popular will eventually have their own life cycle, which has a distinct ending point based on when the software vendors discontinue their product support and move their attention to new product lines. This particular technology evolution in middleware has some dramatic effects on application software because the middleware is closely integrated with many of the emerging application capabilities. When a technology like ActiveX becomes obsolete, it then becomes necessary to upgrade application systems to the new technologies in order to maintain vendor support and integration of new capabilities. The demise of ActiveX can already be seen on the horizon as COM+, a succeeding technology, makes inroads into replacing core elements of its technology. The software interfaces are likely to be quite different, especially because COM and COM+ are based upon an interface definition language, not the same one as CORBA, and COM+ doesn't have an interface definition language, at least in terms of current marketing information. It is important for the software architect to anticipate these kinds of inevitable changes and to plan the migration of application systems to the new technologies in a manner that doesn't mitigate the business purpose of current system development.

The architect is faced with many challenges in the application space. Some of the most strenuous challenges involve the changing business processes that current businesses are undergoing. There is increasing competition from all sectors, and a merger of capabilities through technologies like the Internet, such that newspapers, computer companies, cable television vendors, and telecommunications operators are starting to work in the same competitive spaces and are experiencing significant competitive pressure that is the direct result of information technology innovations and innovative concepts implemented in application systems. Even with previous generations of technologies, it is fairly well known that requirements change a great deal. In fact, the majority of applications costs for software development can be traced directly to requirements changes [Horowitz 1993]. For the first time in history, information technology budgets are exceeding payrolls in many organizations in industries such as financial services. Information technology is becoming synonymous with competitive advantage in many of these domains. However, the basic capabilities of system development are still falling far short of what is needed to realize competitive capabilities fully. For example, in corporate development, one out of three systems that are started end up in a project cancellation [Johnson 1995]. These types of statistics represent inordinate risk for small and medium-size businesses, given the increasing cost and dependence upon information systems.

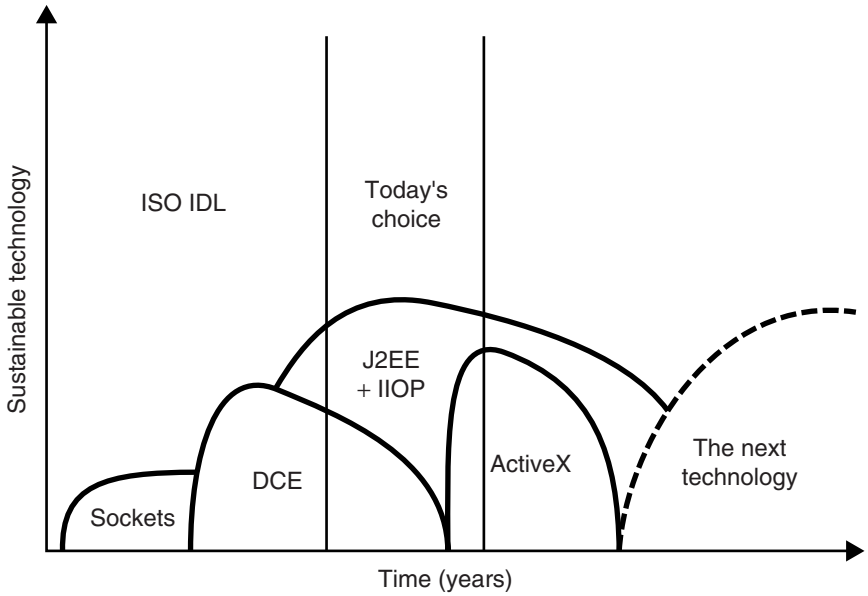

FIGURE 3.16 Managing Technology Change

One of the fundamental rules of thumb of computing is that no technology ever truly goes away. One can imagine some early IBM minicomputers that are still faithfully performing their jobs in various businesses around the world. As information technology evolves, the need to integrate an increasing array of heterogeneous systems and software starts to become a significant challenge. As it becomes necessary to integrate across enterprises and between enterprises using intranets and extranets, the architectural challenges become substantial. One problem is the current inadequacy of information technology infrastructure, including technologies like COM+ and CORBA, which differ from the real application needs in some significant ways. As the challenges of information technology continue to escalate, another problem with the software skill base arises. In many industries, there are substantial shortages of software engineers. It is estimated that at one time there was at least a 10% negative unemployment level in the United States in the software engineering profession. Some industries are much harder hit than others, including public sector systems integration contractors. To build systems with that challenge in mind, the object-oriented architect needs to plan the system development and control the key software boundaries in a more effective manner than has ever been done before.

Many critical challenges lie ahead for application systems developers and software architects. There is an escalating complexity of application system development. This complexity is driven by the increasing heterogeneity of infor-

mation systems and the need to integrate increasing scopes of systems both within and outside the company. In addition, the user requirements not only are increasing the user expectations as a result of exposure to Internet technologies and other marvels of modern life but also are driving software developers to take increasing risks with more complicated and ambitious systems concepts. The key role of the object-oriented architect is the management of change. Managing commercial technology innovation with its many asynchronous product life cycles is one area. Another area is managing the changing business processes that the information technology supports and implements. One area of potential solutions lies in the users influencing the evolution of open systems technologies, influencing software vendors to provide whole technology capabilities, and influencing legislators to put in place the appropriate guarantees of merchantability and fitness for the purposes that underlie the assumptions in system architecture and development.

3.9 APPLYING STANDARDS TO APPLICATION SYSTEMS

In the adoption of object-oriented architectures and technologies, many common questions are raised. They must be resolved to understand the implications fully. The questions of defining object orientation and the component technologies that comprise object technologies have already been reviewed, as has a discussion on how object technologies compare with others, such as procedural technology.

Many other questions and requirements are crucial to certain categories of applications. Questions about performance, reliability, and security on the Internet and how these technologies integrate with vendors that have significant market share are all important considerations in the adoption of these technologies. The next few chapters explain some of the fundamental concepts that describe the commercial and application sides of object-oriented architecture. Furthermore, the case is made for the application of open systems technologies in object-oriented software development practice. Additionally, application development issues on applying object technology, integrating legacy systems, and monitoring the impact of these technologies on procurement and development processes are addressed.

It is important to understand that commercial technologies based upon open systems evolve according to certain underlying principles. These principles are clearly defined through a model developed by Carl Cargill that

describes the five stages of standardization (Figure 3.17). To initiate an open systems standards process, it is necessary to define a reference model. A reference model defines the common principles, concepts, and terminology that are applied across families of standards. These reference models also apply to object-oriented architectures and the integration of application systems. Reference models are an element that is often missing in software engineering processes that address complex issues. Developing a formal reference model through a formal open systems process takes a considerable amount of effort from numerous people.

A typical reference model from the international standards organization may take up to ten years to formulate. Based upon a reference model, a number of industry standards can be initiated and adopted on a somewhat shorter time scale for formal standardization; this ranges up to seven years. Both reference models and industry standards are usually the intellectual product of groups of technology vendors. The standards represent the most general common denominator of technologies across the largest consumer base. In order to apply these technologies, it is necessary to define a number of profiles that serve the role of reducing the complexity of applying the standard within a particular domain or set of application systems (refer to Figure 2.13b).

There are two different kinds of profiles. *Functional profiles* define the application in general terms of a standard for a specific domain. Typical domains might include mortgage lending or automobile manufacturing. The

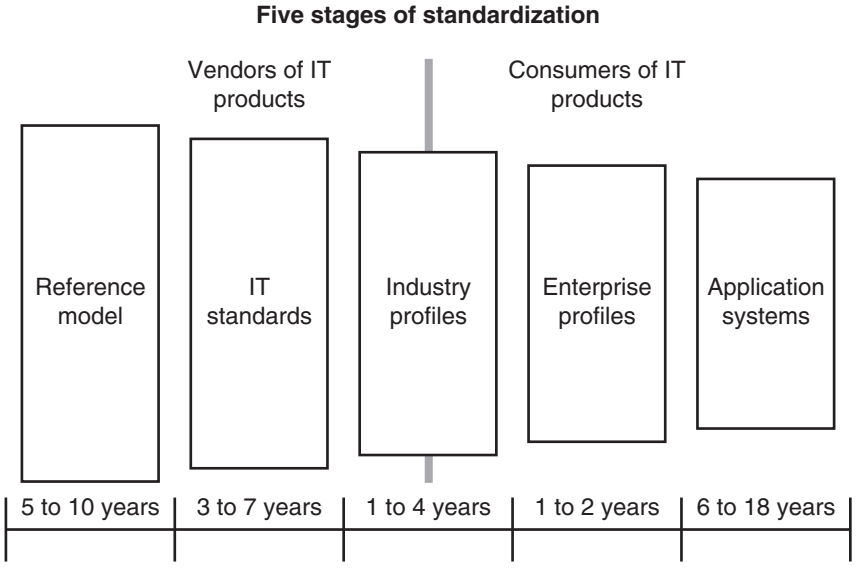

FIGURE 3.17 The Five Stages of Standardization

functional profiles define the common usage conventions across multiple companies within the same industry. Functional profiles can be the product of information technology vendors but usually are a joint product between the users of technology and the vendors.

The next level of profiles is called *system profiles.* System profiles define how a particular family of systems will use a particular standard or set of standards. The family of systems is usually associated with a certain enterprise or virtual enterprise. For example, a set of electronic data interchange standards for the Ford Motor Company defines how the company and its suppliers for the manufacturing process can provide just-in-time inventory control so that Ford's assembly lines can proceed in an organized fashion without interruptions.

Above system profiles are *application systems,* which are specific implementations. Even though the concept of profiles is new to many software engineers, profiles are implemented, perhaps implicitly, in all systems. Whenever a general-purpose standard or a commercial technology is applied, decisions are made regarding the conventions of how that technology is used, and those decisions comprise a profile. Unfortunately, many of the important profiles are buried in the implementation details of information systems. Notice that, in Figure 2.13b, the time scales for developing each of the types of specifications is decreasing. The intention is that the reference models provide a stable architectural framework for all the standards, profiles, and systems that are developed over a longer term. The industry standards provide the next level of stability and continuity, the profiles provide stability and consensus across domains and application families, and all these mechanisms support the rapid creation of application systems on the order of half a year to a year and a half.

Figure 3.18 shows the breakout of reference models and profiles from the perspective of a particular vendor of information technology. In general, a vendor is working from a single reference model that spans a number of industry standards. The vendor implements technologies conformant with these standards and then works with various application developers and vertical markets to define the usage of the technology for valuable business systems. There is a multiplying factor for vendors in this approach in that for a small group of vendors potentially numerous customers are enabled by the technologies that they supply.

Figure 3.19 portrays the concept from the perspective of the end-user application developer. This diagram is somewhat amusing in a dark sense, but it is very representative of the kind of challenges that object-oriented architects in all kinds of information technology are facing today. For a given application system, numerous standards and reference models are potentially applicable to the development of that system. A smaller number of functional profiles and

system profiles can be obtained off the shelf to guide application system development. In general there is a gap between the application implementations and the industry standards in the area of profiling. Because profiling is primarily the responsibility of users, it's appropriate to say that the users are to blame for this gap in guidance.

When profiles are not agreed to between application system projects, the likelihood is that the systems will not be interoperable, even though they are using identical industry standards and even products from the same vendors. This can be a confusing and frustrating situation for application architects. It is necessary to understand these principles in order to resolve these kinds of issues for future system developments.

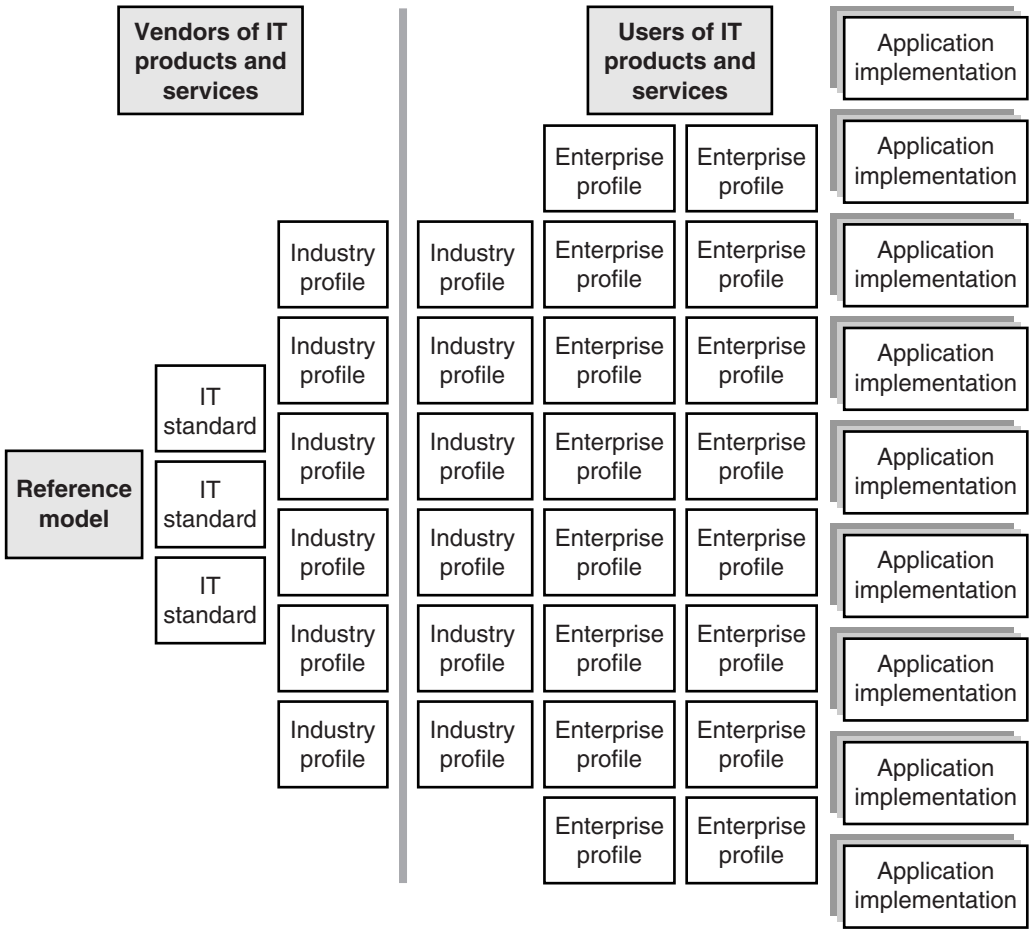

FIGURE 3.18 Standards from the Vendor's Perspective

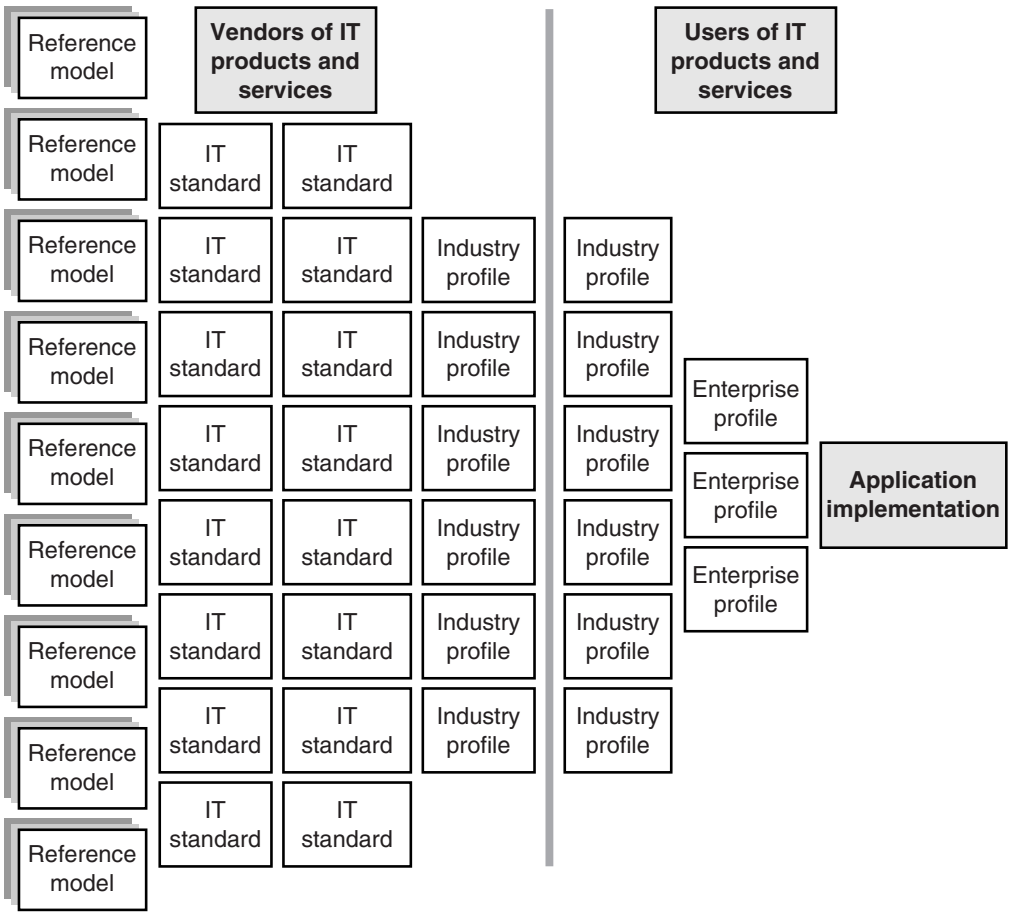

FIGURE 3.19 Standards from the User and Application Developer's Perspective

3.10 DISTRIBUTED INFRASTRUCTURES

Earlier, the concept of middleware was introduced. Middleware provides the software infrastructure over networking hardware for integrating server platforms with computing clients, which may comprise complete platforms in their own right.

Distributed infrastructure is a broad description for the full array of object-oriented and other information technologies from which the software architect can select. Figure 3.20 shows the smorgasbord of technologies available on both client server and middleware operating system platforms [Orfali 1996]. On the client platform, technologies include Internet Web browsers,

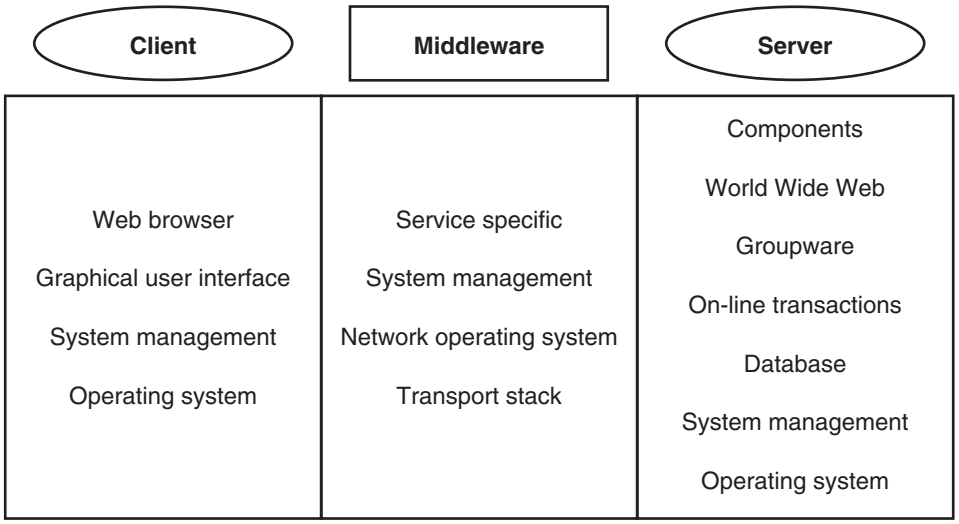

FIGURE 3.20 Infrastructure Reference Model

graphical user interface development capabilities, system management capabilities, and operating systems. On the server platform, there is a similar array of technologies including object services, groupware capabilities, transaction capabilities, and databases. As mentioned before, the server capabilities are migrating to the client platforms as client-server technologies evolve. In the middleware arena, there is also a fairly wide array of client-server capabilities. These include a large selection of different transport stacks, network operating systems, system management environments, and specific services. These technologies are described in significant detail in a book by Bob Orfali, Dan Harkey, and Jeri Edwards, *The Client Server Survival Guide* [Orfali 1996].

Some of the key points to know about client-server technologies include the fact that the important client-server technologies to adopt are the ones that are based upon standards. The great thing about standards is that there are so many to choose from. A typical application portability profile contains over 300 technology standards. This standards profile would be applicable to a typical large-enterprise information policy. Many such profiles have been developed for the U.S. government and for commercial industry. The information technology market is quite large and growing. The object-oriented segment of this market is still relatively small but is beginning to comprise enough of the market so that it is a factor in most application systems environments.

As standards evolve, so do commercial technologies. Standards can take up to seven years for formal adoption but are completed within as short a time as a year and a half within consortia like the OMG. Commercial technologies are evolving at an even greater rate, trending down from a three-year cycle that

characterized technologies in the late 1980s and early 1990s down to 18-month and one-year cycles that characterize technologies today. For example, many vendors are starting to combine the year number with their product names, so that the obsolescence of the technology is obvious every time the program is invoked, and users are becoming increasingly compelled to upgrade their software on a regular yearly basis. Will vendors reduce innovation time to less than one year and perhaps start to bundle the month and year designation with their product names?

The management of compatibilities between product versions is an increasingly difficult challenge, given that end-user enterprises can depend upon hundreds or even thousands of individual product releases within their corporate information technology environments. A typical medium-sized independent software vendor has approximately 200 software vendors that it depends upon in order to deliver products and services, trending up from only about a dozen six years ago. Figure 3.21 shows in more detail how commercial technologies are evolving in the middleware market toward increasing application functionality. Starting with the origins of networking, protocol stacks such as the transmission control protocol (TCP) provide basic capabilities for moving raw data across networks.

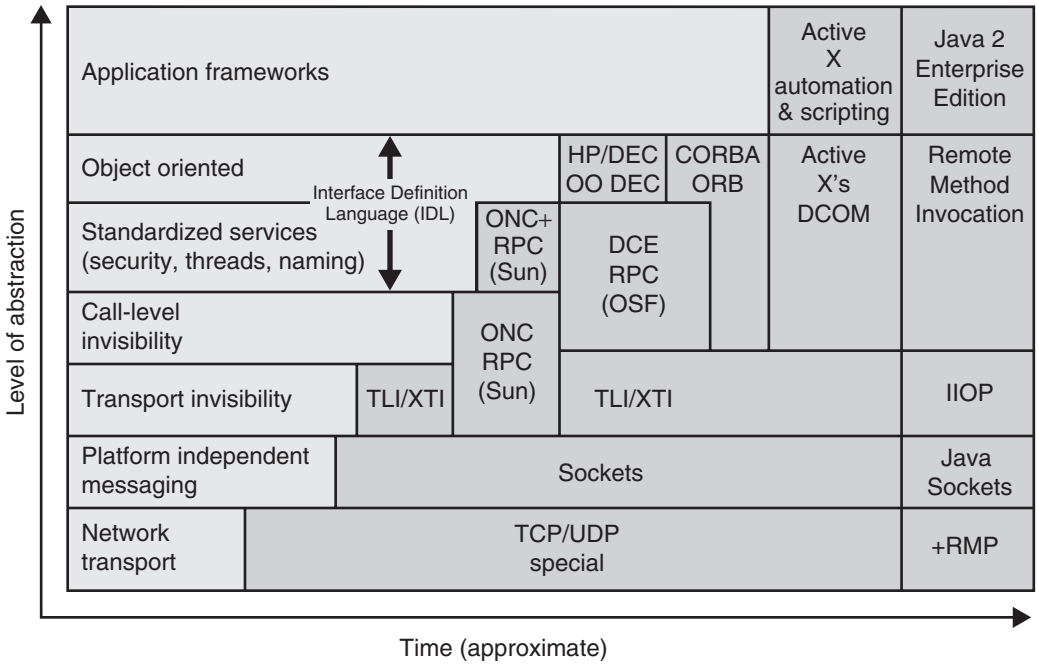

FIGURE 3.21 Evolution of Distributed Computing Technologies

The next level of technologies includes the socket services, which are available on most platforms and underlie many Internet technologies. These socket services resolve differences between platform dependencies. At the next layer, there are service interfaces such as transport-layer independence (TLI), which enables a substitution of multiple socket-level messaging services below application software. As each of these technologies improves upon its predecessors, additional functionality, which would normally be programmed into application software, is embodied in the underlying infrastructure. One consequence of this increasing level of abstraction is a loss of control of the underlying network details in qualities of services that were fully exposed at the more primitive levels. Beyond transport invisibility, the remote-procedure-call technologies then provide a natural high-level-language mechanism for network-based communications. The distributed computing environment represents the culmination of procedural technologies supporting distributed computing. Object-oriented extensions to DCE, including object-oriented DCE and Microsoft COM+, now provide mechanisms for using object-oriented programming languages with these infrastructures.

Finally, the CORBA object request broker abstracts above the remote procedure's mechanisms by unifying the way that object classes are referenced with the way that the individual services are referenced. In other words, the CORBA technology removes yet another level of networking detail, simplifying the references to objects and services within a distributed computing environment. The progress of technology evolution is not necessarily always in a forward direction. Some significant technologies that had architectural benefits did not become successful in the technology market. An example is the OpenDoc technology, which in the opinion of many authorities had architectural benefits that exceeded current technologies like ActiveX and JavaBeans.

Standards groups have highly overlapping memberships, with big companies dominating most forums. Groups come and go with the fashions of technological innovation. Recently Internet forums (W3C, IETF) have dominated, as have JavaSoft and Microsoft open forums.

Many networking and open systems technologies as well as other object-oriented standards are the products of now defunct consortia. The consortium picture is dynamic. Some of the former consortia such as the Open Software Foundation and X Open are now merged to form The Open Group. Other consortia, such as the Object Management Group and the Common Open Software Group, are highly overlapping in membership. A recent addition to the consortium community has been the Active Group. The Active Group is responsible for publishing technology specifications for already released technologies developed by Microsoft (Figure 3.22). The Open Software Foundation originated

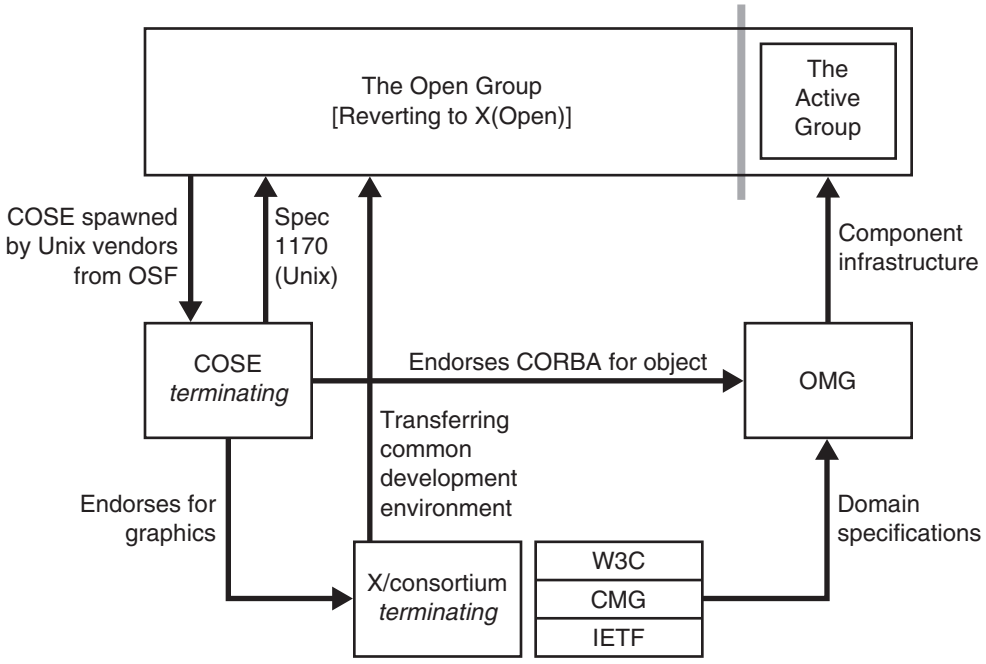

FIGURE 3.22 Commercial Software Technology Consortia

the distributed computing environment that supports remote procedure calls as well as other distributed services. The distributed computing environment is the direct predecessor of the Microsoft COM+ technologies. The distributed computing environment represents the consensus of a consortium of vendors outside Microsoft for procedural distributed computing.

Along with CORBA, the distributed computing environment is a mainstream technology utilized by many large-scale enterprises (Figure 3.23). One important shortcoming of the distributed computing environment is the provision of a single-protocol-stack implementation. As distributed computing technologies evolve, it becomes increasingly necessary to provide multiple network implementations to satisfy various quality-of-service requirements. These requirements may include timeliness of message delivery; performance; and throughput, reliability, security, and other nonfunctional requirements. With a single-protocol-stack implementation, the developers of applications do not have the capability to provide the appropriate levels of service. The technology gap described here is properly described as access transparency, a term defined by an international standards organization reference model that is covered in Chapter 9. Proper object-oriented distributed computing infrastructures do provide access transparency and give developers the freedom to select the appropriate protocol stacks to meet the application quality-of-service requirements.

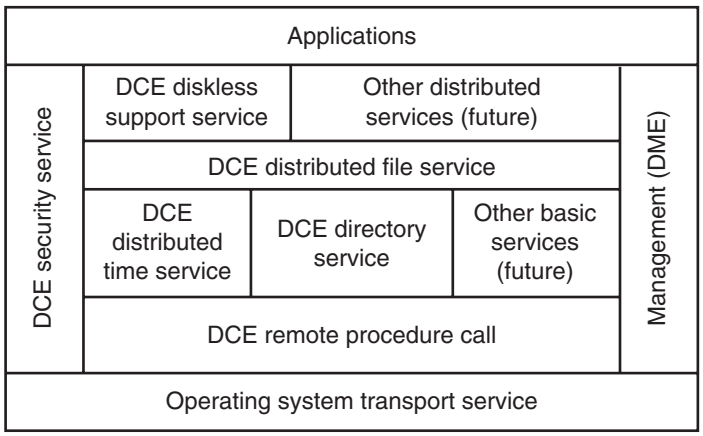

FIGURE 3.23 Distributed Computing Environment

Figure 3.24 shows the infrastructure technologies from the Microsoft COM+ and ActiveX product lines. The basis of these technologies for distributed computing came from the original OSF environment, but that technology was extended in various ways with proprietary interfaces that also support the use of C++ programs in addition to the C program supported by DCE. The ActiveX technologies have a partition between capabilities that support distributed computing and capabilities that are limited to a single desktop. The desktop-specific capabilities include the compound document facilities. Compound document facilities support the integration of data from multiple applications in a single office document. When moving a document from desktop to desktop, there can be complications because of the lack of complete integration with the distributed environment.

Figure 3.25 shows some of the underlying details of how the component object model and COM+ model interface with application software. Application software is exposed to Microsoft-generated function tables that are directly related to the runtime system from Microsoft Visual C++. The consequence of this close coupling between Visual C++ in applications software is that the mapping to other programming languages is not standardized and in some cases is quite awkward (e.g., when ordinary C programs are applied with the COM+ infrastructure). The CORBA technologies provide a resolution of some of these shortcomings.

Figure 3.26 shows the basic concept behind an Object Request Broker (ORB). The purpose for an ORB is to provide communications between different elements of application software. The application software providing a service is represented by an object. This object may encapsulate software that is not object oriented. An application client can request services from an object by

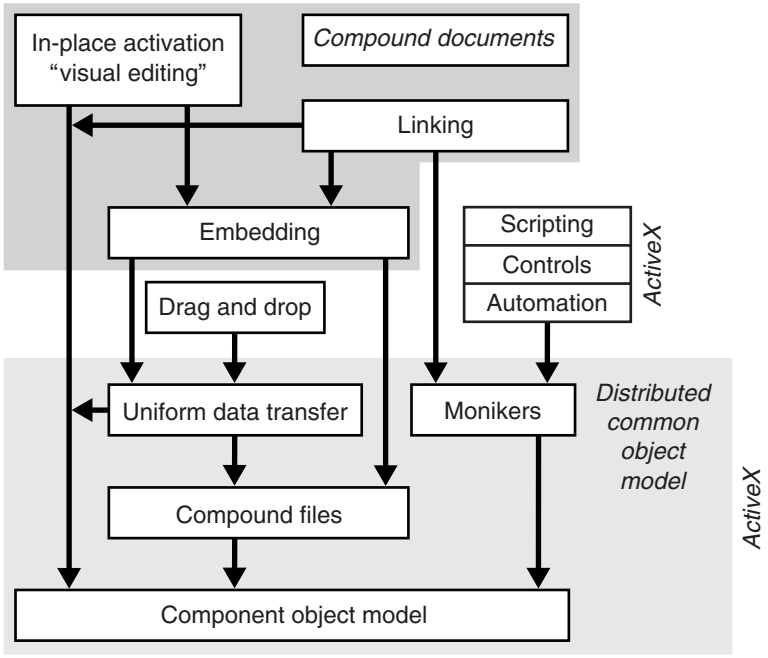

FIGURE 3.24 ActiveX Technology Elements

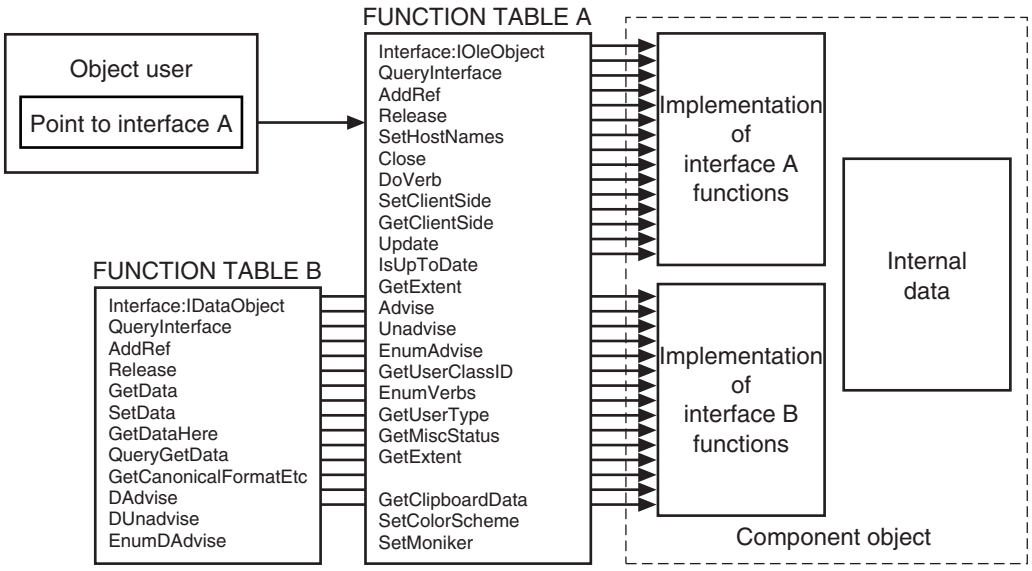

FIGURE 3.25 Component Object Model

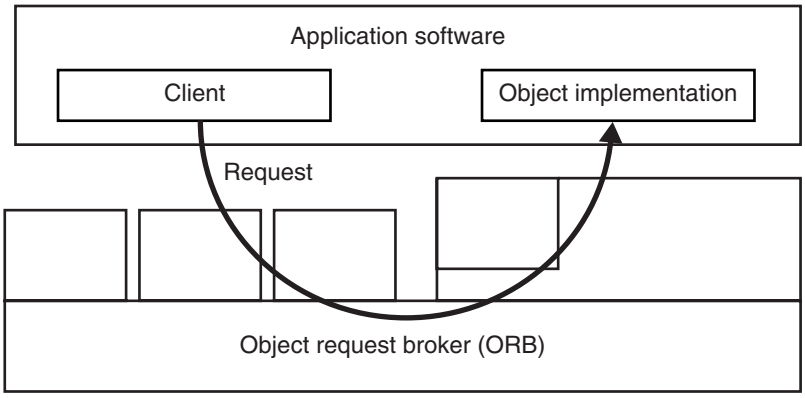

FIGURE 3.26 Object Request Broker Concept

sending the request through the ORB. The CORBA mechanism is defined to help simplify the role of a client within a distributed system. The benefit of this approach is that it reduces the amount of software that needs to be written to create an application client and have it successfully interoperate in a distributed environment.

Figure 3.27 shows some of the finer grained details from the CORBA model. Figure 3.27 relates to Figure 3.26 in that the client and object software interoperate through an ORB infrastructure. The part of the infrastructure standardized by CORBA is limited to the shaded interfaces between the application software and the ORB infrastructure. CORBA does not standardize the underlying mechanisms or protocol stacks. There are both benefits and consequences to this freedom of implementation. Because different implementers have the ability to supply different mechanisms and protocol stacks underneath CORBA interfaces, a number of different products support this standard and provide various qualities of service. Some implementations, in fact, provide dynamic qualities of service that can vary between local and remote types of invocations. The consequence of this freedom of implementation is that the mechanisms selected may not be compatible across different vendors. An additional standard called the Internet Inter ORB Protocol defines how different ORB mechanisms can interoperate transparently. The implementation of IIOP is required for all CORBA products.

The CORBA infrastructure provides two different kinds of mechanisms on both the client and implementation sides of the communication services. On the client side, the client developer has the option of using precompiled stub programs that resemble ordinary calls to the application software. The use of static stubs minimizes the special programming that is required because the

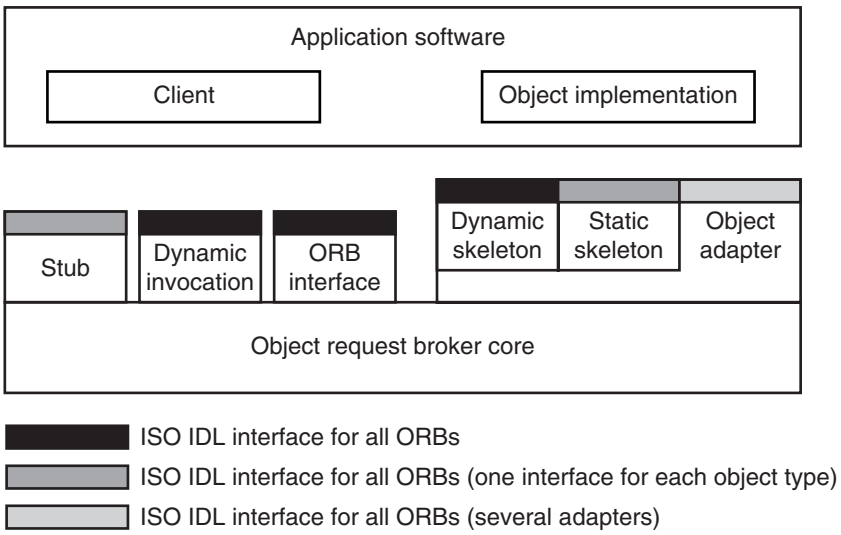

FIGURE 3.27 Key Interfaces in CORBA Architecture

application is potentially distributed. The stub programs appear like local objects in the application environment, but the stubs represent a proxy for the remote object.

The client developer has the option of using dynamic invocation (Figure 3.27). Dynamic invocation is an interface that enables the client to call an arbitrary message invocation upon objects that it discovers dynamically. The dynamic invocation gives the CORBA mechanism extensibility, which is only required in certain kinds of specialty applications. These applications might include program debuggers, mobile agent programs, and operating systems. The implementer of object services in the CORBA environment also has the capability to choose static invocation or dynamic invocation. The two options are generated as either static skeletons or dynamic skeletons.

The skeletons provide the software that interfaces between the ORB's communication infrastructure and the application program, and they do so in a way that is natural to the software developer. By using dynamic skeletons with dynamic invocation in the same program, interesting capabilities are possible. For example, software firewalls, which provide filtering between different groups of applications, can easily be implemented by these two dynamic capabilities.

Figure 3.28 shows the CORBA technologies in the object management architecture and how these technologies relate to the Cargill model discussed

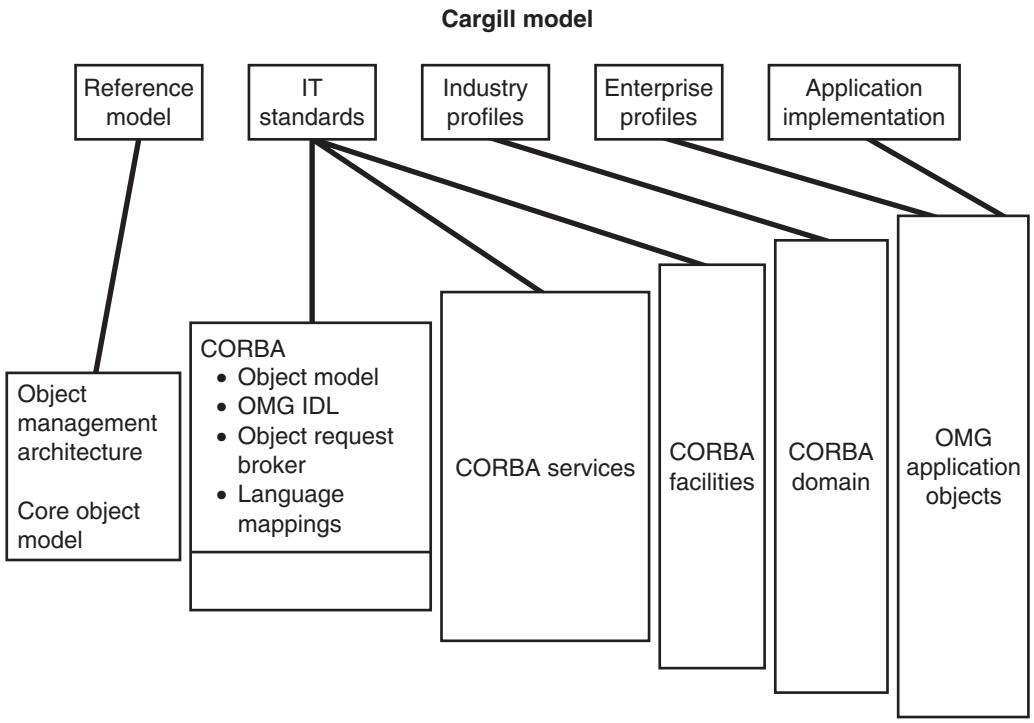

FIGURE 3.28 Extensions of the Object Management Architecture

earlier. The object management architecture shown in Figure 3.9 provides a reference model for all the CORBA technologies. CORBA and the related standards, such as CORBA services and CORBA facilities, are examples of industry standards that apply broadly across multiple domains.

The CORBA domains comprise functional profiles in the Cargill model. In other words, the CORBA domain interface specifications represent domain-specific interoperability conventions for how to use the CORBA technologies to provide interoperability. Finally, the application objects in the object management architecture correspond directly with the application implementations in the Cargill model.

Other initiatives (besides CORBA) have attempted to specify comprehensive standards hierarchies. First Taligent, then IBM's San Francisco project attempted to define object standards frameworks, but neither garnered the expected popularity. Java J2EE has come closest to achieving the vision and represents outstanding progress toward completing the standards picture.

3.11 CONCLUSIONS

This chapter introduced the fundamental concepts of object orientation, open systems, and object-oriented architectures. It also discussed object orientation in terms of isolating changes in software systems by combining the data and processing into modules called objects. Object technology is a capability that is already present and entering the mainstream of software development. Object technology is broadly supported by commercial industry through software vending and by many mainstream end-user organizations in their application development.

As discussed, the only sustainable commercial advances are through open systems forms of commercial technology. With proprietary technologies, the obsolescence of capabilities conflicts with the need to build stable application environments that support the extension of application functionality.

Additionally, stovepipe systems are the pervasive form of application architecture but can be reformed into more effective component object architectures. In the next chapter, object technologies and various reference models that make these technologies understandable will be described.

This chapter considered one of the key concepts in object-oriented architecture—the application of standards in software development. A proper understanding of how standards are utilized is very important to the successful exploitation of commercial technologies and the interoperability of application functions.

In this chapter, object-oriented client-server technologies were described. These technologies focus on the underlying distributed computing capabilities and how they compare with related technologies from the procedural generation. The companies that supply these technologies have highly overlapping interests that are expressed through commercial standards consortia and formal standards bodies. In fact, the distributed computing environments vary from the CORBA mechanism to the Microsoft technologies that are more closely related to remote procedure call. Finally, some of the details of CORBA infrastructure and how they relate to the Cargill model were described.

Also, the different architectural layers, including two-tier, three-tier, *N*-Tier, and peer-to-peer approaches, were examined. The advantages of using different layering techniques were covered and provide essential guidance in deciding when a particular project merits one of the more complex architectural alternatives.

In conclusion, a wide range of open systems client-server technologies support object orientation. These technologies enable the construction of a wide array of distributed systems based upon objects and components.

3.12 EXERCISES

EXERCISE 3.1 Assess the state of your current organization (or customer) with respect to the adoption of software paradigms. Prepare a short status assessment document containing recommendations for resolving any gaps in the current skill base.

BACKGROUND FOR SOLUTION: First look at the programming languages being used. Most procedural and OO organizations adopt single-language solutions. Then examine the training requirements. How much training is each developer required to have? We know of major IT organizations that require 9 weeks to as much as 26 weeks of training before they turn developers loose on the shop floor. At a bare minimum, we would suggest 3 weeks. Suppose we're pursuing the OO paradigm. The recommended training is 1 week for "thinking objects," 1 week for OO programming, and 1 week for distributed systems development process and practice (e.g., experiencing systems building in a training environment). These are the recommended absolute minimums. Some of the smartest companies require much more.

EXERCISE 3.2 Assess the state of architectural control within your organization. Are you heavily dependent upon the architecture of a single vendor or set of vendors? What elements of the architecture do you control in a vendor-independent manner? Create a list of recommendations for resolving any discrepancies or shortcomings resulting from excessive vendor dependency.

BACKGROUND FOR SOLUTION: Ask people, "What is our architecture?" If the answer is Oracle or Microsoft, you should be concerned. These are honorable vendor firms, but in our way of thinking, what vendors do is not application architecture. Simple selection of a technology is not sufficient to resolve architectural forces. At a minimum, your enterprise architecture should describe the deployment of technologies and customization conventions for how products are used consistently across systems development. Ideally, your organization has its own APIs that resolve key interoperability issues, as well as rigorously maintained profiles for technology utilization.

EXERCISE 3.3 Assess the state of middleware technologies in your organization (or customer). Identify which technologies are utilized and how effectively they are exploited.

BACKGROUND FOR SOLUTION: In our experience, there is a very high correlation between the technologies utilized and the architectural practices. If you are using several middleware infrastructures in a single application, you are most likely to have ad hoc architectural practices and relatively unmaintainable systems. In the era of CORBA enlightenment, begin to recognize the folly of this approach. Many organizations, being conservative, chose DCE as their corporate middleware solution. However, DCE remains a relatively brittle infrastructure (originating from the "C" procedural generation of technologies). Early adoptions of CORBA frequently resemble DCE-like solutions. As these organizations mature in their use of distributed computing, there is a corresponding flowering of architectural practices. Eventually, solid architectural frameworks like RM-ODP become quite attractive to these organizations because they help architects think much more effectively about managing infrastructure.

EXERCISE 3.4 Describe a case-study experience for your organization as a useful lesson learned for other developers. Which products, versions, and platforms were utilized? How did you use and customize the applications to meet the needs of the application?

BACKGROUND FOR SOLUTION: A case study or "experience report" is quite different than a design pattern although they both share lessons learned. A case study is a specific instance of a successful solution. As you work through this exercise, think about answering the questions that would be most useful to developers encountering a new architectural problem. What elements of the solution are most reusable, in a way that saves time and eliminates risk for readers about to define a new system architecture?

EXERCISE 3.5 Describe the infrastructure dependencies of one or more current applications in your organization. How would you re-architect these systems in their next generation to accommodate technology change more effectively?

BACKGROUND FOR SOLUTION: The worst case is if you are applying vendor technologies without profiling conventions and user-defined APIs. Unfortunately, the worst case is also typical of most organizations. Suppose a vendor provides 300 APIs to access its product. Your developers will use alternate sets of APIs for each project and even within a single system. If you want to migrate to something else, you have a supreme challenge. Consistency in use of product features can work wonders for enabling interoperability and maintainability. The user-defined APIs, although proprietary, are very much under control and not likely to be vendor specific (e.g., CORBA IDL interfaces). To resolve these issues, you need to simplify the choices for how to utilize vendor products (i.e., using profiles) and clearly identify which aspects will be vendor-

independent. Reliance on standards is one step. Definition of profiles shows that you have sophistication in the use of standards and products.

EXERCISE 3.6 Which standards are being applied in your organization? Do they supply the desired benefits? Are there any profiles for these standards in your organization? Why or why not? Develop a plan, listing the recommended profiles of standards for your organization. Explain the rationale for why your organization needs each profile specification.

BACKGROUND FOR SOLUTION: Standards, while being one step away from vendor dependence, pose many of the same challenges as integrating with vendor-specific APIs. By definition, standards are very general purpose, applying to as many types of applications as possible. Therefore, the management of complexity is not an important goal for the standards writer. In fact, many standards are overly complicated in order to create barriers for vendor competition. Sophisticated application architects know this, and they plan to manage this complexity (e.g., profiles). We apologize for being so singled-minded about profiles, but this is a key solution concept that most organizations miss—with resulting negative consequences. In one of our favorite quotes, a senior executive laments that "We have created a set of fully standards-compliant stovepipes which can't interoperate." It's dead obvious why that's happened. You didn't read our book—not that we created the concept, which is nearly as old as IT standards themselves.

EXERCISE 3.7 Describe the quality-of-service requirements for the distributed infrastructures in your organization (or customer). What qualities of service are readily supported today? What qualities of service could be usefully added? What distributed technologies would be applicable to meet these needs?

BACKGROUND FOR SOLUTION: A quality of service (QoS) is an important category of architectural requirements for distributed infrastructure. Do you need reliable communications (e.g., funds transfer)? Do you need to support continuous media (e.g., desktop video teleconferencing)? How reliable? How continuous? How secure? These are important questions that drive the selection of infrastructures, the migration plans of enterprises, and the practices of enterprise architects.