8

CORBA

A Case Study of an Industry Standard Computing Infrastructure

with Kurt Wallnau

The Object Management Architecture is the architecture for a connected world. Heterogeneous, distributed systems, with worldwide connectivity, is here to stay, and OMA provides the tools to build those systems.
— Richard Mark Soley
Vice President and Technical Director
Object Management Group

This chapter presents object management architecture (OMA) and its better-known communication infrastructure, Common Object Request Broker Architecture (CORBA). Together, they provide a standard software platform in which distributed object-oriented programs can communicate and interact with each other.

Object-oriented technology is not new. Programming languages, such as Simula'67 and its more popular descendant Smalltalk'80, established the core concepts of *object-oriented* software. Programming languages with mass appeal such as C++, ObjectiveC, and most recently Java, have brought these concepts into widespread use. Object technology, as it has come to be known, has made inroads in other areas as well, such as object-oriented database management systems. For some, object orientation has become a philosophy (some might say religion) for software engineering, as evidenced by the use of the term *object-oriented* as a prefix to other well-established terms such as *database management systems*.

When seen in this light, the OMA and CORBA represent just another application of object-oriented concepts to different aspects of computing—in the case of CORBA, to distributed (object-oriented) systems. Thus, an object request broker (ORB) allows objects to publish their interfaces and allows client programs

Note: Kurt Wallnau is a member of the technical staff at the Software Engineering Institute, Carnegie Mellon University.

(and perhaps other objects) to locate them anywhere on the computer network—possibly anywhere in the world—and to request services from these remote objects. Indeed, the initial name for Sun Microsystem's ORB was DOE, for "distributed objects, everywhere."

But CORBA is not an ORB—it is a specification for a *common ORB architecture*. What types of business needs led to the creation of an industry standard architecture for ORBs? How did CORBA address these needs, and how successful is the CORBA design? In what ways has the original need for CORBA changed, and how has CORBA changed in response? These are the questions we consider in this chapter.

8.1 Relationship to the Architecture Business Cycle

The business motivation for development of the OMA comes very clearly from Redmond, Washington, headquarters of Microsoft. Microsoft's success in dominating the personal computing industry throughout the 1980s is well known. Workstation computer manufacturers such as Digital Equipment Corporation, Hewlett-Packard (HP), and Sun Microsystem initially watched the deadly competition among PC manufacturers and among Microsoft, IBM, and Apple with detachment. Toward the end of the 1980s, however, this detachment was gradually replaced by genuine alarm: The price–performance ratio for PCs was improving, and PC processing power was getting close to that of high-end workstations. The hardware distinction between PCs and high-end workstations was disappearing. Workstation manufacturers realized that the competitive fray that they had so far avoided was about to overtake them.

THE FIGHT AGAINST MICROSOFT

Because profit margins for PC hardware are small, hardware is only a small part of the story. Control of the operating system leads to control of the vast market of independent software vendors (ISVs) and their wares. Microsoft's Windows operating system now dominates the PC marketplace. Further, extensions to Windows, such as object linking and embedding (OLE), allow ISVs to develop components—stand-alone applications with a common look and feel—that can be readily integrated into a standard desktop environment. OLE also allows documents (data files) produced by components to be embedded within the documents of other components, essentially supporting the composition of components. The possibility that OLE offers of seamless integration among components such as spreadsheets, word processors, report writers, databases, and so on, was very attractive to end users.

The major computer manufacturers had not, singly, been able to meet these expectations nor had they contributed to the development of the diverse and burgeoning ISV and component marketplace supported by Microsoft Windows. Fortunately for computer manufacturers, however, PCs and OLE were hobbled by

the DOS lineage of Microsoft Windows. Although adequate for personal use, Microsoft Windows would not scale, or be sufficiently reliable, for the highly demanding and critical information-management needs of corporate users. The computing manufacturers' best assets were their stable, mature operating systems and the enterprise software built to run in these environments. End users were willing to endure an impoverished, expensive, and unintegrated component marketplace to have the security of stable and mature operating systems.

Microsoft's Windows NT appeared to be the first major threat to this uneasy equilibrium. Unlike Microsoft Windows, Windows NT is an operating system with a solid pedigree. With the PC price–performance ratio not just converging but leveling with that of high-end workstations and "servers," with an advanced operating system to exploit cheap PC hardware, and with OLE and a vast, ready-to-hand component marketplace, it became clear that computer manufacturers needed to unite to provide adequate competition.

OPEN STANDARDS: AN EMBARRASSMENT OF RICHES

The major computer manufacturers had not been idle. One of the major problems facing computer manufacturers in their battle with Microsoft was market and technology fragmentation. Consider, for example, the number of variants of the UNIX operating system: Solaris, SunOS, SystemV, HP/UX, and so forth. Historically, there have simply been too many versions of UNIX (let alone other operating systems such as VMS, MVS, etc.) running on too many different hardware platforms to provide a unified front against Microsoft. To counteract this fragmentation, manufacturers and ISVs with a stake in their non-Microsoft products have spawned numerous industry groups to develop an "open standards" software infrastructure that can remove or accommodate heterogeneity.

Unfortunately, the number of industry groups and the open standards they supported also reflected the diversity of technology and business objectives lurking just beneath the surface. Rivalry between the major vendors (IBM, HP, Digital, Sun) was always apparent. A plethora of organizations and acronyms promoted by one or several of the major vendors assaulted the end user: Open Systems Foundation (OSF), X/Open, distributed computing environment (DCE), POSIX, OpenDoc, Motif, OpenLook, common desktop environment (CDE), common operating system environment (COSE), Berkley/UNIX, and SystemV/UNIX, were a few of these organizations and systems. This assault increased the appearance of the very fragmentation that these organizations and standards were intended to remedy in the first place. Moreover, the collaborators on these open standards are also competitors in their own right. As a result, standards were often augmented with value-added features as a means of obtaining favorable product differentiation in the marketplace.

But product differentiation runs counter to the needs of ISVs who want and need a uniform computing environment in the face of heterogeneous technology, and the diversity and complexity of choices posed by the open systems standards was a liability.

THE OBJECT MANAGEMENT GROUP

The Object Management Group (OMG) was chartered in 1989—not coincidentally just one year after Microsoft established its Windows NT design team—to "create a component-based software marketplace." Of course, such a marketplace already existed, but it was "owned" by Microsoft. The OMG envisioned a technology whereby applications (components) could execute on different platforms and operating systems, yet still interoperate. This would be an alternative to Microsoft's OLE. But why would OMG succeed where others had failed? What unifying principles could OMG use that could capture the attention of ISVs and unify the disparate complexity of open systems computing?

In a word, the answer is *objects*. In three words, it is object management architecture, OMG's proposed answer to Microsoft hegemony. Figure 8.1 shows the architecture business cycle (ABC) as it pertains to the OMA and CORBA. The customers for OMA and CORBA were hardware and software vendors such as IBM, Digital, and Sun. The end users were the ISVs. The technical environment at the time consisted of OLE (the competition) and object-oriented everything (design, analysis, databases, and so on). The architects were members of a consortium, so they had a variety of backgrounds.

The quality attributes for the OMA and CORBA are discussed in the next section; they include buildability, balanced specificity, implementation transparency, interoperability, evolvability, and extensibility. The architects were the OMG itself, and the final systems were all of the systems to be constructed based on CORBA and the OMA.

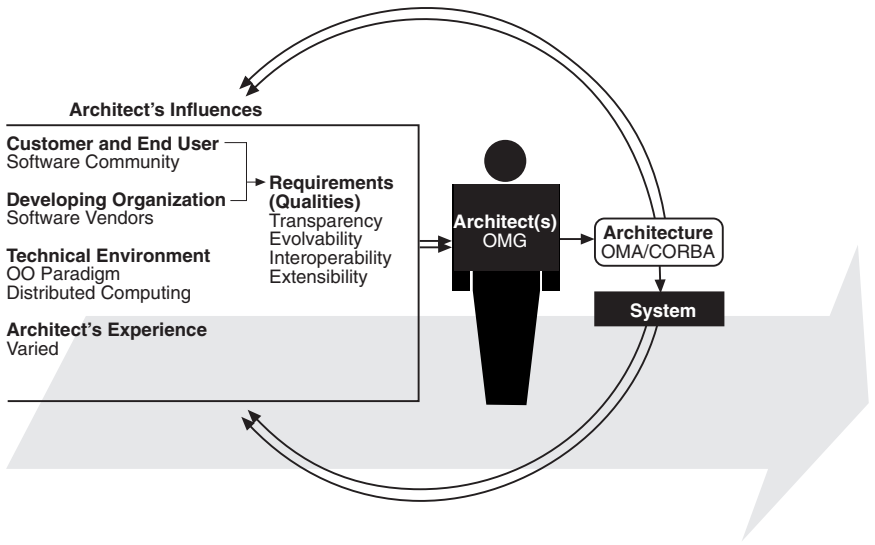

FIGURE 8.1 The ABC as it pertains to OMA and CORBA.

8.2 Requirements and Qualities

This section discusses the goals of the OMG and how these goals were reflected in qualities for the OMA.

NECESSARY CONDITIONS FOR THE SUCCESS OF THE OMG

The idea underlying the OMG is that a component industry that is independent of a particular vendor's infrastructure (i.e., Windows NT and OLE) can be established, provided that

- A component integration infrastructure can be developed that is sufficiently rich in services to support most component integration requirements
- Implementations of these services are readily available on all significant platforms (including Wintel machines, that is, Windows running on Intel processors)
- Clients are portable across vendor implementations of these services and can interoperate with components that use implementations of these services provided by differing vendors.

In addition to these positive success conditions, the OMG also had to avoid numerous pitfalls. These are best illustrated by the failure of the portable common tools environment (PCTE). In Chapter 1, we discussed how systems can influence the environment in which future systems are developed. This influence can be both positive and negative. PCTE is an example of a negative influence.

PCTE was intended to provide standard services for component integration, and yet despite the hundreds of millions of dollars lavished on it, PCTE never succeeded in the marketplace. Although the ultimate cause of the PCTE failure can be argued, the failure does underscore a number of potential pitfalls:

- PCTE was conceived as an integrated (monolithic) system, which meant that the designers had to fully specify all of its technology services and how these services interact (e.g., security, distribution, data, and process management services).
- The complexity of PCTE and the fact that subset implementations of PCTE were not allowed resulted in steep development costs, lengthy time to market, and ultimately few PCTE implementations in the marketplace.
- The PCTE specification required the use of (what was then) immature "research" technologies, for example, distributed object-based data management.
- Producing a formal International Standards Organization (ISO) standard is a lengthy process—PCTE development began in the early 1980s, but it did not become an ISO standard until 1994. By the time it became a standard, its technology assumptions were largely obsolete.

TABLE 8.1 OMA Quality Attribute Requirements

Goal	How achieved
Buildability	Be readily implemented with available technology, across different computing platforms, and work well with existing software development tools and practices
Balanced specificity	Be detailed enough to provide a meaningful standard for component developers and integrators, but general enough to allow vendor-specific features and optimizations
Implementation transparency	Provide compete transparency of implementation details so that client programs can be independent of object implementation details (programming language, object location, operating system, vendor, etc.)
Interoperability	Support interoperation of objects implemented on different vendor implementations of the OMA and allow bridges for interoperability of the OMA to other technologies (including Microsoft technology)
Evolvability	Be responsive to the development of new object and distributed-systems technologies and new market needs (i.e., be open to new services and facilities and accommodate interoperation of many kinds of object technology)
Extensibility	Provide a stable, core set of component-integration and interoperation services needed by most component developers, but be extensible for component integration services needed by market niches.

The business conditions and the lessons from PCTE are reflected in the OMA's desired quality attributes for their specified architecture, as summarized in Table 8.1.

It is interesting to note what is *not* a *required* quality attribute for the OMA. Neither security nor performance is required since the OMA has to balance numerous market factors if it is to become a ubiquitous component-integration standard, as it must to compete with Windows NT/OLE. Each quality attribute implies a set of engineering trade-offs, and each trade-off has consequences that might be anathema to the OMG in certain market segments. For example, rigorous performance requirements might not be acceptable to SmallTalk implementations of the OMA. On the other hand, high-performance implementations of the OMA might be developed by vendors seeking market distinction. Similar vendor differentiations can emerge (and have emerged) to support other quality attributes, such as fault tolerance.

In addition to the challenge of deciding which quality attributes to address and which to leave as vendor enhancements, the OMG also had to develop an architecture that made reasonable trade-offs among the selected quality attributes. For example, buildability is in tension with evolvability; as we will discuss, complete client-side implementation transparency and the generality requirements of OMA specifications (the second clause in balanced specificity) are in tension with each other; and both are in tension with the ORB interoperability requirement. Thus, the OMG needed to ensure that the OMA addressed a wide array of trade-offs—an architecture that is detailed versus general, stable versus flexible, featureful versus implementable, general purpose versus niche optimal, niche optimal versus interoperable, and so on.

8.3 Architectural Approach

The OMG approach to satisfying these quality attributes is through two architectures: the OMA and CORBA. The OMA is the high-level architecture, and its structure reflects quality attributes that relate to extensibility and evolvability. It also reflects the consensus-based design processes inherent in any standards-making efforts and in the OMG design processes in particular. CORBA is an architecture for one component of the OMA, and it reflects the deeper technical requirements that relate to buildability, implementation transparency, and interoperability. Both OMA and CORBA reflect balanced specificity.

The OMA describes what it means to be an object, how objects communicate with each other, and how applications can be made to interoperate through the use of standard object services. Figure 8.2 depicts a high-level view of the OMA. The OMA assumes three different types of objects: objects that provide universally useful services (CORBAServices), objects that provide useful but not universally useful services (CORBAFacilities), and objects that provide application-specific services (application objects). What differentiates these types of objects is the kinds of services they provide; however, they are all objects. In the OMA, all objects communicate via an ORB; CORBA defines a standard architecture (and services) required of OMA-compliant ORBs.

The partitioning scheme outlined in Figure 8.2 is important despite the fact that there are no objective criteria for determining whether an object is a CORBAService, CORBAFacility, or application object. Instead, the partitioning scheme is meant to separate the kinds of services that are needed by most developers (CORBAServices) from those that are more likely to be niche specific (CORBAFacilities) are those likely to be used by a single application) and from the application itself (application objects). All of these are separated from the services that are necessary to enable object communication—the ORB.

A summary description of CORBAServices (as of October, 1996) is provided in Table 8.2. The CORBAFacilities category is not as mature and is instead

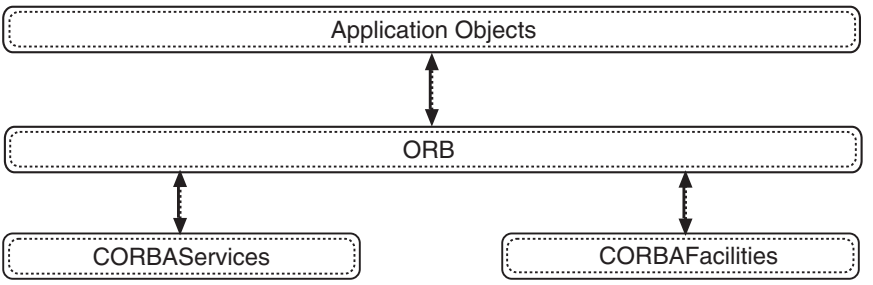

FIGURE 8.2 The object management architecture.

TABLE 8.2 Summary of CORBAServices

Service	Service description
Naming	Provides the ability to bind a name to an object; similar to other forms of directory service
Event	Supports asynchronous message-based communication among objects
Life-cycle	Defines conventions for creating, deleting, copying, and moving objects
Persistent-object	Provides a means for retaining and managing the persistent state of objects
Transaction	Supports multiple transaction models, including mandatory "flat" and optional "nested" transactions
Concurrency-control	Supports concurrent, coordinated access to objects from multiple clients
Relationship	Supports the specification, creation, and maintenance of relationships among objects
Externalization	Defines protocols and conventions for externalizing and internalizing objects across processes and across ORBs

focused on further partitioning of CORBAFacilities into smaller categories organized into "horizontal" facilities such as information, system, task, and user-interface management and "vertical" facilities targeted for particular application domains such as imagery, manufacturing, distributed simulation, accounting, oil and gas exploration, and mapping. Figure 8.3 illustrates how the different classes of objects in the OMA are related to each other. CORBAServices are the most fundamental application building blocks (for using the OMA); CORBAFacilities reuse and

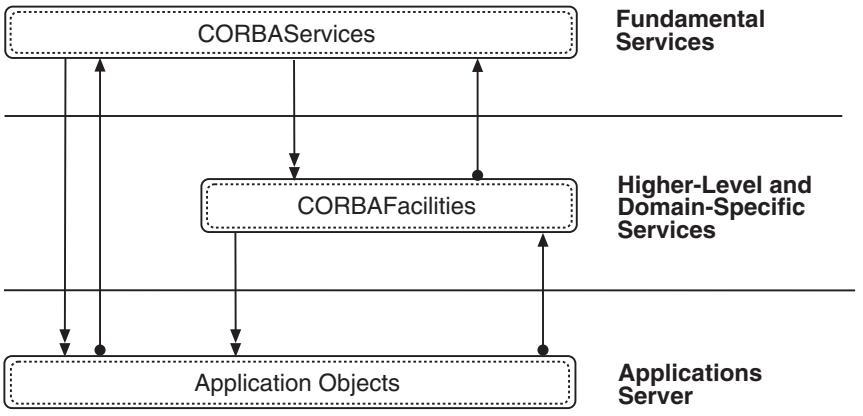

FIGURE 8.3 Contextual relationships among OMG object classes.

extend these building blocks and introduce domain-specific building blocks; application objects reuse and extend both CORBAServices and CORBAFacilities. However, application objects are not a subject for standardization in the OMA.

The partitioning scheme depicted in Figures 8.2 and 8.3 is important for two reasons. First, it provides a different forum for stakeholders who have separate, and sometimes competing, interests. For example, although the OMA is intended to be an infrastructure for component developers, the ORB is really the infrastructure of the infrastructure—it is the communications heart of the OMA. As such, its specification is the most complex part of the OMA, and its implementors are most likely to be the computer manufacturers or ISVs in the CORBA business. In contrast, CORBAFacilities provides a forum for component developers and integrators in specific market niches such as manufacturing (the domain-specific services alluded to in Figure 8.3). CORBAServices are somewhere between these two extremes, and the specification of these services draws accordingly from both manufacturers and ISVs.

A second reason for the OMA partitioning is that it supports several quality attributes. First, the core ORB part is separated from questions concerning the functionality that should be supported by the OMA (such as outlined in Table 8.2). Consequently, ORBs could (and did) appear on the marketplace long before the specification of CORBAServices was complete. Thus, one of the pitfalls of the PCTE—the long time to market—was avoided. Further, the separation of functional services from the ORB set in motion an incremental specification and implementation for the OMA itself. Thus, CORBA (the ORB specification) became an initial stable basis for the OMA, CORBAServices provided commonly needed services, and the CORBAFacilities provided a basis for developing niche-specific services. Finally, the extensible nature of these services—new services can be added more or less independently of each other—supports the ability of the OMA to accommodate new technologies, although as we will see, CORBA also supports this quality attribute.

In addition to the structure and content of the OMA itself, the OMG's process for developing the OMA also contributes to satisfying the OMA quality attributes. The byzantine nature of this development process defies a brief description. The process is also as flexible and evolvable as the OMA itself (i.e., it continues to change). However, the process is one of consensus and compromise, balanced with a clear bias toward already implemented solutions to proposed extensions of the OMA.

Table 8.3 summarizes how the structure of the OMA and the processes of the OMG help to satisfy the quality attributes outlined in Table 8.1. As can be seen, quality attributes are achieved through a combination of technical and nontechnical means. On the technical side, the use of an abstract interface description and the functional partitioning scheme help achieve *balanced* specificity, evolvability, and extensibility. On the nontechnical side, the OMG standardization process helps achieve buildability. Thus, what the OMA depicted in Figure 8.2 lacks in specificity, it makes up for in the harmonization of business considerations (market niches, addressing stakeholder interests, and so on).

TABLE 8.3 How the OMA Supports Its Quality Attribute Requirements

Goal	How achieved
Buildability	OMA standardization process combines a competitive "request-for-proposal" approach for developing specifications with a requirement that selected specifications have commercial quality implementations available within a year of acceptance.
Balanced specificity	Specifications described via an abstract interface definition language (IDL). (See CORBA discussion in Section 8.4 for more details about IDL.)
Evolvability	Specification is partitioned into separately evolvable subcategories. OMA has processes to introduce task forces and special-interest groups.
Extensibility	ORB core and CORBAServices form a stable core. CORBAFacilities allow niche-specific extensions.

8.4 Architectural Solution

The ORB provides the fundamental communications services between clients and objects in the OMA. The ORB is responsible for all of the mechanisms required to find the object implementation for a client request, to prepare the object implementation to receive the request, and to communicate the data making up the request. The interface that the client sees is completely independent of the location of the object, the programming language in which the object is implemented, and anything else that is not reflected in the object's interface.

CORBA specifies an architecture for OMA-compliant ORBs. This section focuses on CORBA v2.0, released in July of 1995. However, before getting into the details, it is worthwhile relating the story of CORBA v1.0 and why it was designed as it was. As noted, the OMG issues requests for proposals for specifying OMA technologies. The original CORBA specification was no different, and numerous responses to the OMG's request were submitted. The major computer manufacturers submitted their proposals, but the team led by Sun Microsystems prevailed. Business requirements forced an alternative proposal, submitted by a team led by Digital, to be also incorporated. Certainly it would not satisfy the OMG's objectives to create a schism at this early juncture.

The OMG solution? Combine the Sun and Digital proposals so that no major players would walk away losers. The influence of the original Digital submission can be seen in the CORBA dynamic invocation interface (DII), while the original Sun submission emphasized static invocation via "stubs" and provided the overall structure of what became CORBA. A careful scrutiny of the CORBA specification today will reveal some subtle inconsistencies between the DII and stub-based paradigms. However, this compromise, at the cost of a small degree of impurity, produced harmony in the OMG and resulted in a specification that reflected the interests of most major computer manufacturers.

COMPONENTS OF CORBA

History aside, the overall structure of CORBA is depicted in Figure 8.4, which corresponds (without client and object implementation) to the ORB component of the OMA (see Figure 8.2). Figure 8.4 shows the structure as the OMG presents it. Figure 8.5 shows the structure of a typical CORBA implementation, recast into our notation. In our discussion of CORBA, we will concentrate on how this structure, and the functionality it partitions, achieves the required OMA quality attributes.

First, though, a bit of terminology is in order. Figure 8.4 differentiates client from object implementation. A *client* is a computer program that makes requests of services provided by an object implementation. Of course, an object implementation (or just *object* where this is not confusing) can be a client of other objects. Thus, it is also often useful to distinguish between clients, which can be application programs or objects, and *pure clients*, which are not objects and provide no services that can be accessed via an ORB.

Both clients and object implementations use the same ORB interface (the same for ORBs from any vendor) to do ORB bookkeeping such as binding to an ORB and binding to an object implementation. The individual clients and object implementations could use either a static interface where the interface is described at specification time and does not change for the life of the execution or a dynamic interface where the client dynamically determines the details of the

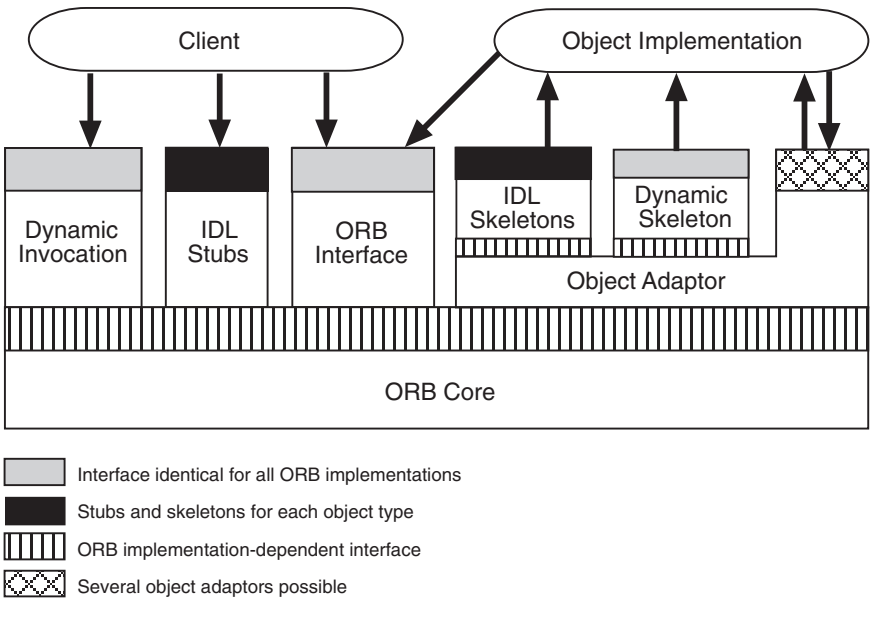

FIGURE 8.4 The structure of CORBA-compliant ORBs as presented by OMG.

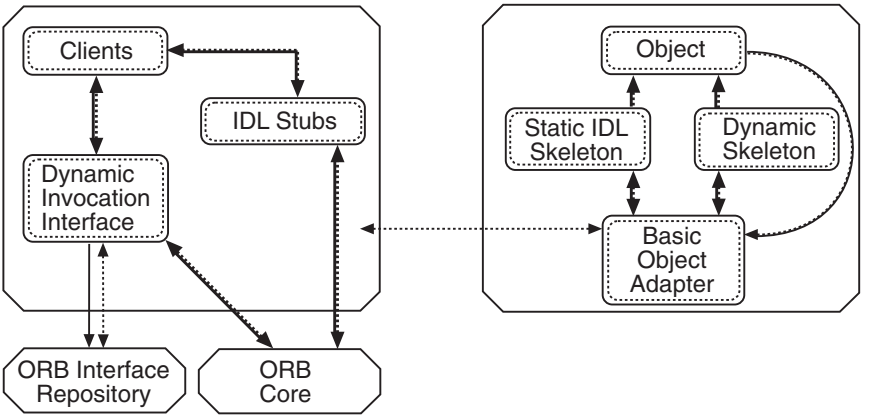

FIGURE 8.5 An example CORBA-compliant implementation.

object implementation's interface. In this discussion, we will focus on the static interface description for simplicity.

Objects in the OMA are described via the OMG's interface description language (IDL). As its name suggests, IDL describes the interface, *not* implementation details, to objects. In fact, this is an important property because clients of objects may assume nothing more about objects than is specified by their interface and perhaps narrative prose concerning the "semantics" of an operation defined in an IDL. A sample IDL is illustrated in Figure 8.6. Although the banking abstraction illustrated is not realistic, it illustrates a number of interesting points. First, C++ programmers will be instinctively familiar with IDL. Replace the IDL keyword `interface` with the C++ keyword `class` and remove the IDL keywords `readonly` and `attribute` (`attribute` is just syntactic sugar anyway), and the example would be legal C++ code.

Why is this important? Recall that one of the OMA quality attributes was that it must be readily implementable with available technology and work well with existing development technology (*buildability*). Selecting a C++-like syntax for IDL made the task of generating C++ code from IDL simpler than it would have been had a more abstract IDL syntax been chosen. Also, the C++ flavor satisfies the principle of least surprise (i.e., don't introduce new concepts where old ones will do). Since C++ programmers constitute the largest object-oriented developer market, meeting them at least halfway was appropriate.

On the other hand, IDL is also considerably more restrictive in interface description capability than is C++. For example, most C++ programmers would overload the account-creation operations defined in `Bank` to be just `open_account` or something similar. Why does IDL not allow overloading? Because overloading is not supported by all programming languages (e.g., C, Fortran). Supporting overloading

```
interface Account {
       readonly attribute string name;
       readonly attribute float balance;
       void deposit (in float amount);
       void withdraw (in float amount);
};

interface CheckingAccount: Account {
       readonly attribute float overdraft_limit;
       void order_new_checks();
};

interface SavingsAccount: Account {
       float annual_interest();
};

interface Bank {
       CheckingAccount open_checking(
              in string name, in float starting_balance);
       SavingsAccount open_savings(
              in string name, in float starting_balance);
};
```

FIGURE 8.6 IDL for a simple bank abstraction.

in IDL would interfere with the objective of making CORBA work with heterogeneous technologies (i.e., *buildability* and *interoperability*).

Of course, it would still be possible to support overloading in IDL and to then define "name-mangling" approaches for generating nonoverloaded interfaces in languages such as C. However, although this might produce a more elegant and more object-oriented IDL, it would make the IDL translators more complex, make them more costly to develop, and perhaps have a negative effect on both time to market and on the development of IDL processors for non-object-oriented languages. This, in turn, would interfere with OMG's objective of getting OMA implementations rapidly into the marketplace for as diverse a set of platforms and technologies as possible (*buildability*).

Note that IDL is used to specify the interfaces described in Figure 8.4, in particular the ORB interface, dynamic invocation interface, dynamic skeleton interface, and interfaces to the basic object adaptor and interface repository (neither of which is explicit in Figure 8.4). IDL is also used to describe the interfaces of CORBAServices and, when they become sufficiently mature, CORBAFacilities. The use of IDL as a means for describing the OMA helps define interface functionality while leaving implementation details unspecified; this is useful for ORB developers (*balanced specificity*). This is also useful for application developers,

whether they are developing clients or object implementations, since they are insulated from ORB implementation details, and further, clients are insulated from implementation details of object implementations (*implementation transparency*).

IDL completely defines the interfaces of objects in the OMA. The IDL stubs and skeleton components illustrated in Figure 8.4 are generated from IDL interface specifications. Thus, the banking interface illustrated in Figure 8.6 would be "compiled" by an IDL compiler that generates stubs to be used by clients and skeletons to be used by object implementations. The IDL mapping for a programming language defines the rules for performing this code generation. Thus, C++ client programs would access banking services defined in IDL in the same way, regardless of which vendor's IDL compiler was used. Standard IDL mappings are defined for C, C++, and Smalltalk; mappings for other languages such as Ada 95 have also been defined and are under review.

To illustrate the idea of standard mapping, Figure 8.7 shows a small client program written in C++ that makes use of IDL stubs generated from the specification outlined in Figure 8.6.

The main points to note are as follows:

- In theory, this code will be the same regardless of which vendor's ORB (and IDL compiler) and which C++ compiler are used. Thus, ISVs should be able to write applications that are portable across platforms (at least insofar as using the OMA is concerned).
- The C++ code corresponds quite closely stylistically and syntactically to the corresponding IDL interface definition. In contrast, other language mappings are not quite so direct (although compromises in IDL expressivity ensure that mapping to conventional, non-object-oriented languages is still quite feasible).
- There is nothing special about the source of objects in the OMA—they are returned from ordinary object implementations such as a naming service or a bank. Clients need no vendor "magic" for creating these fundamental units of computation and distribution.

Thus, the insulation of clients and objects from ORB-vendor-specific interfaces, the use of IDL to describe standard interfaces to various components of CORBA (interface repository, ORB, dynamic invocation interface, dynamic skeleton

```
main (int argc, char **argv){
       Bank *PNB = ...// look this up from the naming service
       CheckingAccount *my_checking =
              PNB->open_checking("John Doe", (float) 15,000);
       my_checking->order_new_checks();
}
```

FIGURE 8.7 Sample client-side code for bank abstraction.

TABLE 8.4 How CORBA Supports OMA Quality Attribute Requirements

Goal	How achieved
Buildability	IDL compilers are easily implemented with conventional parser-generator technology.
	The IDL language mappings to C and C++ mesh well with existing software-development practices.
Balanced specificity	CORBA interfaces are defined in IDL, effectively deferring implementation decisions to ORB implementors.
Implementation transparency	Clients can rely only on properties of objects that can be expressed in IDL, effectively providing transparency of implementation details.

interface), and the use of standard mappings from IDL to popular programming languages help the OMA satisfy quality attributes concerning buildability, balanced specificity, and implementation transparency. Table 8.4 summarizes the key points from the preceding discussion.

OBJECT ADAPTORS AND THE BASIC OBJECT ADAPTOR

The use of IDL allows clients and object implementors to access ORB services, but this is only part of what is needed. In addition to providing a means for object implementations to access ORB functions, some means must be established for the ORB to access object implementations. Specifically, the ORB must be able to *activate* object implementations in response to requests from clients and, once activated, the ORB must be able to *deliver* those requests.

There are many ways in which ORB vendors may implement the object model—what an object is, how objects are activated, how they are executed—and this is no trivial matter. Consider the following three alternatives:

1. Use a commercial object-oriented database management system to implement objects, and use the database management system as a runtime environment for executing object methods.
2. Use a shared-library approach to implement objects, where the actual methods are resident with client programs.
3. Use a traditional remote procedure call (RPC) model to implement objects, where objects are implemented as server processes.

Given the OMG's desire to allow implementation latitude (*balanced specificity*), CORBA should allow each of these forms of implementation in a way that still maintains implementation transparency. Unfortunately, the means by which objects are located and activated and by which their methods are executed differ radically among the three alternatives listed above. Because the ORB core must make assumptions about how objects are implemented, there is an implementation dependency between the ORB and the type of technology used to implement

the CORBA object model. The developers of object implementations (object providers or application developers) have a similar dependency.

So how should the needs of vendors to be able to exploit existing object technologies and support newly emerging object technologies (*evolvability*) be balanced with the need to provide a stable basis for ISVs to develop object implementations that are portable from ORB to ORB across different vendor ORBs (*implementation transparency*)? The CORBA answer to this question is the concept of object adaptor (refer to Figure 8.4) and the requirement that CORBA-compliant ORBs provide an implementation of a particular kind of object adaptor, known as the basic object adaptor. This is an example of what we call a *wrapper* in Section 15.3. Object adaptors provide the following services to object implementations:

- Generation and interpretation of object references
- Mapping of object references to the corresponding object implementations
- Activation and deactivation of object implementations
- Invocation of methods on object implementations
- Registration of object implementations
- Security

The first four items involve the concept of *object reference*, the means by which clients access the services defined (in IDL) for particular objects. Object references are similar to pointers in conventional programming languages, except that they contain more information than pointers. This additional information enables the ORB to perform functions such as locating and activating object implementations. By having a specialized component in the architecture that can generate and interpret object references and by making the form of object reference opaque to all other components in the architecture (including clients), CORBA allows different object-implementation technologies to be "plugged into" an ORB (*extensibility* and *evolvability*).

But object adaptors cover only the ORB-vendor half of the story. The other half of the story—a uniform approach for developing object implementations—is provided by the basic object adaptor. The *basic object adaptor* supports an implementation approach that is quite traditional in modern operating systems and reflects an RPC model. The basic object adaptor equates object implementations with programs that act as object servers. An *object server* (i.e., a program, typically executed as an operating-system process) can be started per method, a separate program per object, or one program for an arbitrary number of objects. The basic object adaptor is easily implemented on all "real" operating systems.

Figure 8.8 illustrates the role of object adaptors and the basic object adaptor in supporting both ORB extensibility to alternative object technology and in supporting (to some extent) uniformity of interfaces for object implementations. The basic object adaptor provides support for object technologies that are implemented using conventional means provided by most operating systems. Thus, it supports a technical approach and development style that is quite similar to RPC,

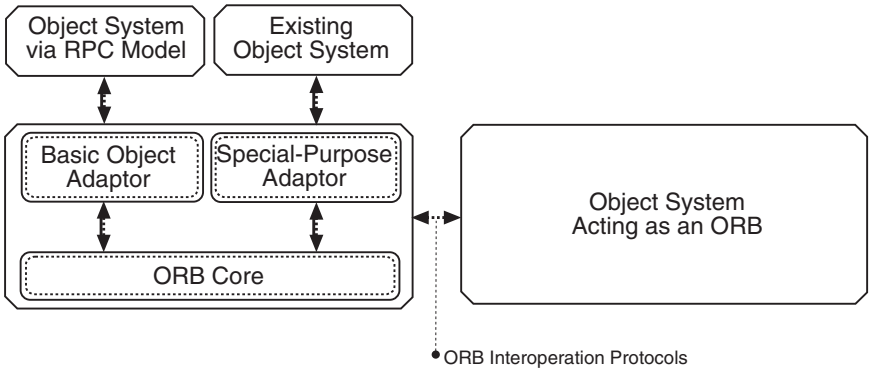

FIGURE 8.8 Object adaptors and integration of alternative object technologies.

is easy to implement using existing and proven technologies, and is standard across all ORBs; this satisfies the buildability quality attribute. Special-purpose adaptors (as illustrated in Figure 8.8) can be developed for other forms of object technology; for example, adaptors for object-oriented database management systems might require database key information in object references, might handle object activation and deactivation differently than the basic object adaptor, etc. The general concept of an object adaptor directly supports the need to allow vendor optimizations and exploitation of new object technologies (*evolvability*).

Figure 8.8 also indicates a third way of introducing new object technologies into CORBA: through interoperation protocols, in this case treating a "foreign" object system as a kind of ORB. The topic of CORBA's interoperation architecture—one aspect of the approach CORBA takes to the question of supporting technology heterogeneity—is considered in the following sections on the CORBA interoperability architecture and interworking architecture. Table 8.5 summarizes the ways in which object adaptors and the basic object adaptor contribute to OMA quality attributes.

CORBA'S INTEROPERABILITY ARCHITECTURE

The ability of clients (pure clients and object implementations that make requests of other objects) to make requests of objects implemented using different vendor ORBs is supported through the CORBA interoperability architecture. It is interesting to note that although interoperability among ORBs is essential to achieving the broadest objectives of the OMA—to provide an open, multivendor infrastructure for component integration to compete with Microsoft's single-vendor solution—the initial CORBA specification did not specify inter-ORB interoperation. It was not until 1995 that the CORBA v2.0 specification dealt directly with the question

TABLE 8.5 How Object Adaptors Support OMA Quality Attribute Requirements

Goal	How achieved
Buildability	The basic object adaptor, in conjunction with the use of IDL stub/skeletons, reflects an RPC approach to the underlying technology and development practices. This approach is mature, implementable on all "real" operating systems, widely used, and well understood.
Balanced specificity	Vendors can develop object adaptors that optimize specific platform capabilities, because the details of which object adaptor is used by an object are transparent to clients of the object. The basic object adaptor itself is not so rigidly specified as to disallow implementation optimizations, for example, optimizations that exploit operating system support for multi-threaded servers.
Implementation transparency	Details of the object adaptor used by an object are transparent to clients of the object.
Evolvability and extensibility	New or alternative object technologies can be integrated with an ORB through the development of special adaptors.

of interoperation. Why the delay? Because inter-ORB interoperation required a degree of vendor cooperation that might have delayed the initial introduction of CORBA. The OMG decided that it was better to get CORBA implementations into the marketplace first and worry about making ORBs interoperable later.

The OMG approach to achieving interoperability was to introduce yet another architecture, the CORBA interoperability architecture. Although what was done in CORBA v2.0 to allow interoperability is not an architecture according to our definition, the OMG calls it an interoperability architecture, and this architecture is based on a number of design goals that augment some of the OMA quality attributes already outlined in Table 8.1. We outline these augmenting quality attributes, taken almost verbatim from the CORBA v2.0 specification, in Table 8.6. Some design goals and quality attributes pertaining to the interoperability architecture (e.g., architecture neutrality, generality, widespread availability) repeat aspects of the OMA requirements and business context already covered in Table 8.1.

TABLE 8.6 CORBA Interoperability Architecture Quality Attributes

Goal	How achieved
Interoperability	Support interoperation of objects implemented on different vendor implementations of the OMA and allow bridges for interoperability of the OMA to other technologies (including Microsoft technology)
Good performance to footprint ratio	Allow high-performance, small-footprint, lightweight interoperability solutions
Backward compatibility	Work with the CORBA-compliant core features of existing ORB implementations
Uniformity	Support all operations implied by the CORBA object model (i.e., no distinction among native and foreign objects in terms of CORBA-defined operations on object references)

The CORBA interoperability architecture uses the concept of bridges that we discuss in Section 15.3. The basic approach is shown in Figure 8.9. Both ORBs in the figure are CORBA compliant, but they do not use the same internal protocols with respect to some concept that they wish to share. The two ORBs communicate to a bridge that provides translation between the internal protocols. Three options are possible, as follows:

1. The bridge is separate as shown in the figure. The bridge is specifically constructed to translate between the protocols of ORB 1 and ORB 2. This leads to a plethora of such bridges and also increases the communication costs, but it has the advantage that neither ORB must be modified to provide interoperate.
2. The bridge is a portion of one of the ORBs, say ORB 1. That is, there is a control as well as a data connection between ORB 1 and the bridge. In this case, it is called a full bridge. ORB 1 is responsible for ensuring that it follows the protocol of ORB 2. This leads to a large number of different protocols that must be supported to ensure interoperability, but it is the most efficient if two vendors wish to cooperate to allow interoperation.
3. The bridge is divided in two between the proxy objects and each half is a portion of one of the ORBs. That is, the bridge is separated, each half is connected to its adjacent ORB using a control and data connection, and the two halves are connected using a data connect. This scheme is called a half-bridge. The two half-bridges communicate using an OMA-defined protocol.

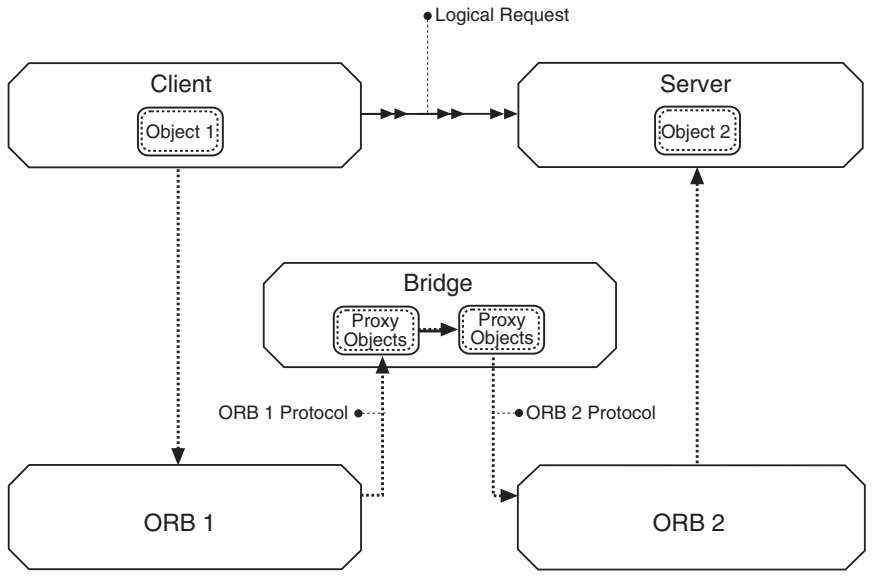

FIGURE 8.9 Use of bridges to achieve CORBA interoperability.

Another way of saying this is that each ORB must support an OMA-defined protocol and is responsible for translating messages received in the OMA-defined protocol into its internal protocol.

8.5 The Web and CORBA

We have seen how the business needs associated with competing with Microsoft and the desire to avoid repeating the experience of PCTE have led to an architecture designed for program extensibility and how the technical aspects of the OMA and CORBA contribute to satisfying the various quality attributes. Evaluating how successful the OMA has been in the marketplace is a different question. Has it been widely adopted, and does it provide an alternative to Microsoft?

One measure, of course, is the number of CORBA-compliant products (i.e., ORBs) that have appeared in the marketplace. Using this measure, the OMA appears to be quite successful, as the following (partial) list of vendors demonstrates: BBN, Digital Equipment, DNS Technologies, ParcPlace, Expersoft, HP, Sun Microsystem, IBM, Iona, and Visigenics. CORBA ORBs are also being developed by university research and development projects, for example, at Stanford, XeroxPARC, and Cornell.

Even more compelling than the number of ORB products is evidence of the use of ORBs in commercial-grade applications (i.e., not just research projects and throwaway prototypes). There is, indeed, indirect evidence of the use of CORBA in "real" application domains. For example, manufacturing, health care, and telecommunications domain task forces have been established by the OMG in response to market demand for CORBA solutions in these different domains. More direct evidence can be found on the home pages of various ORB vendors as a way of advertising their products through testimonials. Iona claims more than a few large-scale uses of their ORB on projects including satellite systems, health care, banking, and trading.

The natural question to ask about the OMA is whether it can somehow exploit the phenomenal growth of Web-based technology, including Java. Alternatively, it is reasonable to ask whether the Web and Java are on the verge of making the OMA obsolete. We take up these questions in turn.

EXPLOITING JAVA AND THE WORLD WIDE WEB

If it is true that mobile and distributed objects are complementary, and that the Web is already a delivery vehicle for mobile objects (via Java), how can the OMA join in? In effect, can the OMA draft in the wake of Java much in the same way that race car drivers and cyclists draft behind their respective front runners? The answer, it turns out, is Yes, and the structure of CORBA has proven resilient enough and flexible to accommodate this. The following paragraphs describe how.

First, we start with the basic notion of a Java "applet," a Java-based computer program that is downloaded from a Web server to a Web browser running at

a client location. Applets provide an interface to browsers, which can then make "back" connections to the source of the applet Java code via low-level Transmission Control Protocol/Internet Protocol (TCP/IP) interfaces. What if, instead of using sockets directly, there were some way to access host services at a higher level of abstraction, using CORBA objects instead of sockets? In effect, the Java mobile code would represent a detachable client interface to immobile CORBA objects. The client interface could be a graphical user interface (GUI), or it could perform smart processing on the client machine. In any event, it makes sense to keep some objects remote—maybe they consume too many resources to relocate, or maybe there are administrative or security reasons for not relocating them—while having other objects that are capable of being relocated. It turns out that this scenario is readily implemented using existing browser technology, Java implementations, and CORBA v2.0.

One approach is to generate Java stubs from IDL interface specifications. However, this is insufficient, because stubs typically link in additional ORB code to perform connection management and all the low-level details of interacting with a remote object. One solution is for ORB developers to write their client-side libraries in Java, too, and to download these libraries to the browser at runtime. This is the solution taken by Sun Microsystems' JOE (Java objects everywhere) system. Another nice feature is that if, like JOE, the client-side libraries use the CORBA-defined ORB interoperation protocol, the CORBA-using applet will be able to interact with any CORBA object, from any vendor, anywhere (subject to security restrictions imposed by the browser itself).

Another approach is for the browser to come bundled with an on-board ORB. Thus, given an object reference in the form of a string downloaded to the browser using the conventional HTTP protocol, the on-board ORB could convert the type of the object being referred to and make requests of the remote object. Details about what operations are supported by an object could be acquired by the browser through queries to the remote object's interface repository. Requests could then be made of the object using dynamic invocation. Netscape has announced that the next major release will feature an on-board ORB, so this also seems to be a viable solution to the development of CORBA using applets.

IS CORBA OBSOLETE?

Although Web-based delivery of CORBA objects (or, rather, access to CORBA objects) demonstrates that the OMA was sufficiently flexible to accommodate rapidly emerging changes in distributed computing technology (one of the key OMA quality attributes), CORBA is no longer the only viable distributed-object technology.

For example, Java now supports remote method invocation (RMI), which in effect builds Java-specific ORB functionality into every Java virtual machine. Because Java's RMI is specific to Java, it is more convenient for Java programmers to use than CORBA. For example, interfaces to remote objects can be expressed with Java's own interface definition notation, rather than in IDL, which

reflects compromises already described. JavaBeans is a related development, which in many ways mirrors the OMA services built on top of CORBA.

Microsoft has not been idle while its competitors, OMA and CORBA, were being defined and implemented. Microsoft's Distributed Component Object Model (DCOM) supports the distribution of objects. It therefore represents a more direct Microsoft-specific competitor to CORBA for developing distributed Wintel systems. Nor has Microsoft ignored the Java angle. By providing its own Web-browser technology and integration scheme for tying together Java, OLE, browsers, and the like, Microsoft has planted itself on firm footing for competing in the distributed-object technology marketplace.

It is doubtful, however, that the OMA will go quietly into the night. As we have shown in this chapter, the OMA is complex, reflects numerous compromises, and is showing signs of age. However, the qualities that have led to its initial successes—openness, vendor freedom, client-implementation transparency—may serve it well as it struggles for new life in a changing environment.

8.6 Summary

The OMA architecture was developed to serve the business needs of the OMG. These business needs were influenced by the lessons of PCTE, by the competitive pressures of Microsoft, and by the requirement to satisfy the major corporate sponsors of the OMG.

The architecture features a separate infrastructure and an expanding list of services. The quality goals of interoperability and extensibility led to the use of IDL as an integration mechanism. The OMA continues to evolve in response to new competitive pressures, but its basic architectural framework has held up over many years.

8.7 For Further Reading

Much OMG information is available on line. This includes the Object Management Group home page, available at *http://www.omg.org*. The Distributed Software Technology Center in Australia also has a home page with interesting OMG information. It is available at *http://corbanet.dstc.edu.au*.

How CORBA treats security was described by Deng and colleagues [Deng 95], and a comparison of CORBA and OLE was published by Jell and Stal [Jell 95]. Other interesting articles on the OMG and CORBA can be found in *Object Magazine* [Roy 95], [Watson 96]. Hamilton [Hamilton 96] discusses the relationship between Java and CORBA.

8.8 Discussion Questions

1. The discussion of CORBA in this chapter centers around programmatic integration of applications. The discussion of the World Wide Web (WWW) in the previous chapter centered around data integration of applications. What is the major difference between these two approaches? What are the strengths and weaknesses of each? What would be involved in moving CORBA toward a datacentric view or the WWW to a programmatic view?

2. What are the issues involved in using CORBA to obtain real-time performance? To put it another way, what would be involved in maintaining real-time quality of service in a distributed object world (as was required in many of the case studies in this book)?