

Figure 4.20. System extension using the four architecture patterns.

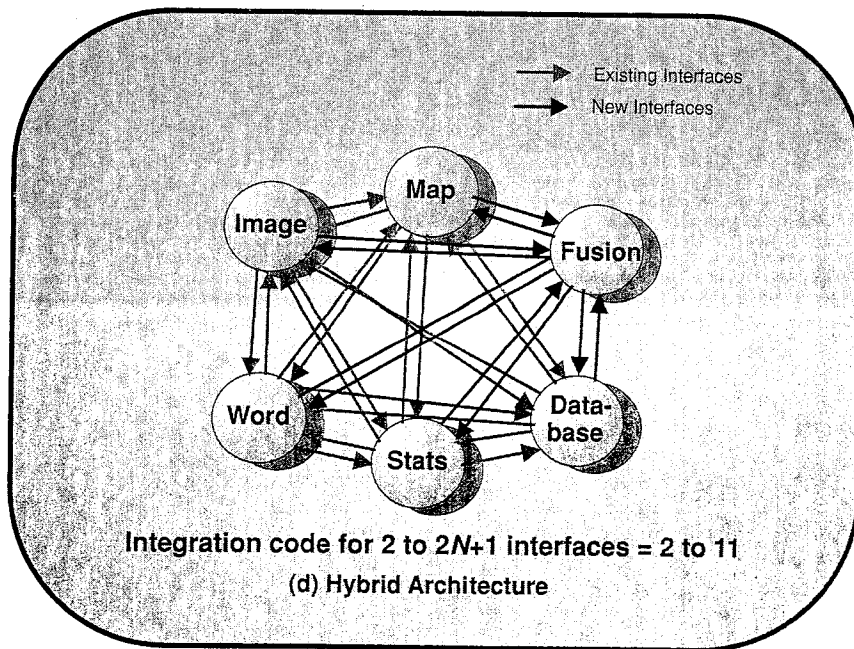
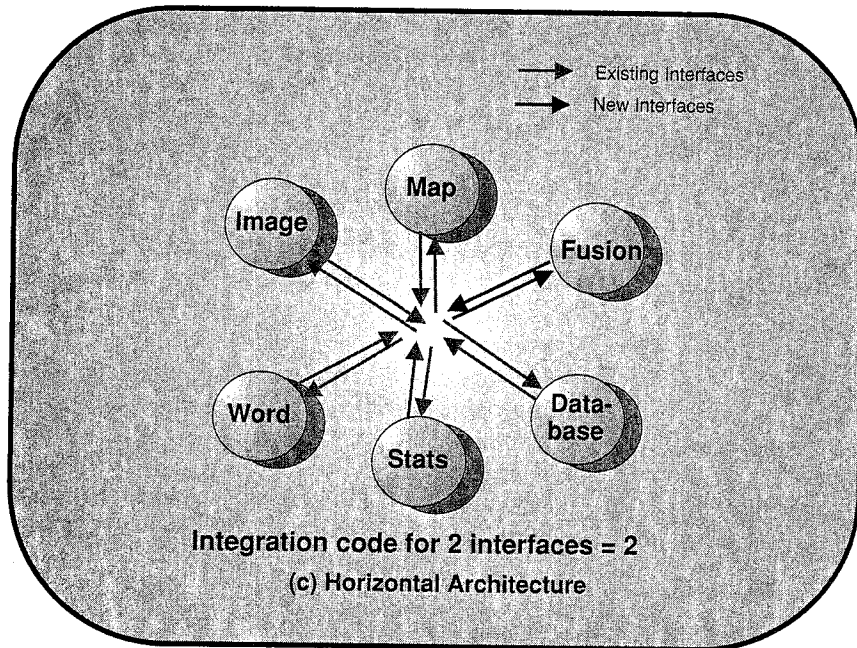


Figure 4.20. continued.

create the system. The second scenario is a system extension. In each case, a sixth application is added to the system. The new application obeys the same architecture pattern conventions. The third scenario is an application replacement. A new application is integrated in place of the one in the original architecture. In the fourth scenario, the two systems are merged to create a common interoperable system with twice as many applications.

In the analysis, the development cost is incurred for all of the integration code needed to implement the architecture-level interfaces. This includes both client and server codes for a particular application-to-application connection. If architecture standards allow the reuse of integration code, then this is reflected as a cost savings.

Figure 4.20 shows the results of the analysis for an arbitrary size system, where  $N$  is the number of applications. This model shows that the custom architecture requires high costs for initial system development and all forms of system modification. The vertical architecture provides no real cost benefits for initial development and system extension. It saves about half the cost of component replacement, and it is cost neutral to system

	#1 Custom	#2 Horizontal	#3 Vertical	#4 Hybrid
Baseline	$N \times N$	$2N$	$N \times N$	$2N$ to $N \times N$
Extend	$2N+1$	2	$2N+1$	2 to $2N+1$
Replace	$2N-1$	2	$N-1$	2 to $N-1$
Merge	$2N \times N + N$	No New I/F	No New I/F	No New I/F

*Table entries show the number of interface implementations needed to integrate the system... multiplies the cost and complexity of the implemented system.  
N = Number of Applications in Integrated System*

Figure 4.21. Comparison of architecture patterns.

	#1 Custom	#2 Horizontal	#3 Vertical	#4 Hybrid
Baseline	-	71 %	0%	36 %
Extend	-	87 %	0%	44 %
Replace	-	86 %	57 %	71 %
Merge	-	95 %	95 %	95 %

Figure 4.22. Example of cost savings  $N = 7$

merger. This indicates that the vertical standards provide for intersystem interoperability (a capability also shared by the horizontal and hybrid architectures). The horizontal architecture provides the most ease of extensibility. The cost of system extension is constant regardless of the size of the system. As we stated earlier, the horizontal architecture provides many benefits, but it is not practical to assume that a community of developers will be strictly limited to common horizontal interfaces. The hybrid architecture includes a range of costs between the horizontal and vertical architecture examples.

System extension is likely to cost at the  $2N + 1$  level of the custom architecture in three out of four cases. The key lesson is that architectural benefits are very sensitive to the way that the system is designed and extended. Potential benefits are significant, but they are likely to be jeopardized by subsequent system extensions which are not conformant to the architectural vision and detailed tradeoffs made by the architect.

Figure 4.22 shows the relative cost savings with respect to the custom architecture when  $N = 7$ , a typical system size. The hybrid can provide the cost benefits of the horizontal architecture with the specialized functional-



ity benefits of the vertical architecture. Average cost savings for the hybrid architecture are substantial for development and all forms of system modification. In summary, the hybrid architecture is our recommended basic pattern for software architectures.

### **Advanced Architecture Pattern: Separation of Facilities**

An advanced architecture strategy involves the separation of facilities. The Internet services have successfully demonstrated this approach. The Internet facilities are partitioned between the front-end tools (the user facilities) and the back-end information sources (the application facilities). The information sources are strictly back-end servers that can be shared by multiple clients. The user facilities are strictly front-end applications that serve one user. Some client applications can access multiple types of servers. The user has a choice of front-end styles for viewing data.

The Internet communication standards are defined in terms of lower-level communication layers, typically at the Internet Protocol (IP) level. Compared to CORBA, the IP level requires a substantial amount of client and server software in support of each interface. CORBA may have future impact on the Internet because it enables the rapid creation of many new services and customized interfaces.

The Internet provides an important example of how to structure an architecture in a distributed environment (see figure 4.23 on page 101). Shared data and services should be migrated toward the back-end application facilities. Customized user functionality should reside in separate user interface applications. Shared functions and data do not belong in the user facilities, and user interaction should not be associated with the shared services.

Beyond the Internet, there are many other reasons why architectures should be partitioned in this manner. In general, architectures should minimize the software investment in the user facilities and migrate as much functionality and data as possible to shared application facilities (see figure 4.24 on page 102). In the following paragraphs we make some of the key arguments why this strategy is necessary.

Shared services such as data repositories and search engines often require special processing resources. Repositories may have massive storage requirements and need high-speed disk interfaces. Computation-intensive services may need special processors to provide adequate response-time performance. For a shared service involving many users, it is usually cost effective to provide specialized processors, buses, and storage systems, tailored to the needs of these services. The cost of special hardware can be distributed among many users.

Most user interface facilities are platform-specific. Applications do not port easily between platforms unless some form of cross-platform development environment is being used. To take advantage of the best capabili-

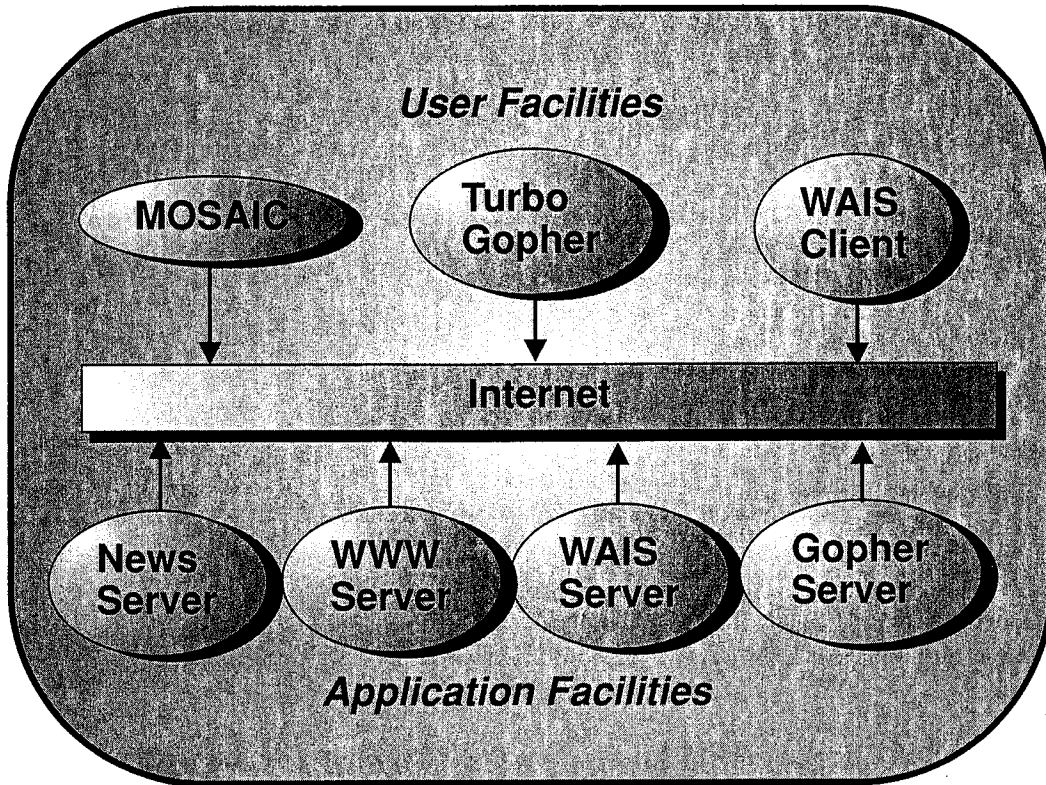


Figure 4.23. Partitioning of Internet.

ties of each platform, it is necessary to write the user interface code in a platform-dependent manner. Platform dependence will continue to be an issue with user interface software because the industry desktop market will continue to become increasingly fragmented. Emerging technologies such as COSE/CDE, OLE/COM, OpenDoc, OPENSTEP, Fresco, and Taligent are environments that will drive the need for partitioned software.

Architects should minimize software investments in platform-dependent software. This can be done by migrating as much functionality as possible to shared application facilities. Dependent software is expensive to port between platforms; the quantity of this type of software should be minimized. Any vendor-dependent software also is in danger of premature obsolescence and high operation and maintenance costs.

Simplifying the role of user facilities allows the creation of more customized user interfaces. It is very difficult or impossible to create a common user interface that is readily accepted by all user groups. The groupware software market is well aware of this issue. The failure of many group

	Partitioning Criteria	Benefits
<b>User Facilities</b>	<ul style="list-style-type: none"> <li>● Front End - Only</li> <li>● End-User Specific</li> <li>● Workflow Dependent</li> <li>● Platform Specific</li> <li>● GUI/UIMS Tool Specific</li> </ul>	<ul style="list-style-type: none"> <li>● User Customization</li> <li>● Isolate Dependent Code</li> <li>● Use Platform Features</li> </ul>
<b>Application Facilities</b>	<ul style="list-style-type: none"> <li>● Back End - Only</li> <li>● Processing Service</li> <li>● Data Source</li> <li>● Memory Intensive</li> <li>● Compute Intensive</li> <li>● Portable</li> </ul>	<ul style="list-style-type: none"> <li>● Sharing Data</li> <li>● Sharing Services</li> <li>● Relocation</li> <li>● Replication</li> <li>● Multiple Delivery Platforms</li> </ul>

Figure 4.24. Separation of user facilities from shared application facilities.

scheduling tools is an example of how user interfaces are closely tied to individual user preferences and group culture. By simplifying the role of user facilities, more customized user interfaces can be created to meet the needs of special-interest users' groups. Simplified user interface facilities require less code and are less expensive and faster to build and specialize. They also adapt to end-user needs and changing requirements.

The emerging market of user-interface rapid prototyping tools will enable dramatic reductions in the cost of user-interface development. Many of these tools will interface directly with CORBA. Independent software vendors such as Bluestone, Integrated Computer Solutions, Oberon, Netlinks, and others are developing or already have CORBA products on the market. It may soon be possible to implement a complete customized user interface within hours. This interface might provide all of the desired CORBA invocations without substantial development. As platforms and user needs change, new user interfaces can be created just as easily.

Some application domains may require an additional category of separated facilities. Workflow represents the organization's business process. An automated workflow supports the business process across multiple software packages. Workflow software can perform functions such as data transfer of results, automatic manipulation of data using multiple packages, and user notification.

In general, shared application facilities should be completely independent of workflow. The shared application facilities should be among the most stable subsystems in the architecture and should support flexibly many modes of utilization eliminating any hard-coded workflow from the access protocols. In some environments, workflow is naturally merged with the user-interface facilities. In others, the user interface facilities need to be workflow independent. This is often the case where there are multiple workflows in an organization that share common applications. Workflow can provide useful animation to a set of passive user facilities and shared application facilities. It is a good example of why good functionality through architecture-level APIs must be provided. A workflow process may need to control virtually any type of function that a software package provides. Workflow can be highly volatile, as an organization adapts to meet changing business needs. For this reason it is probably prudent to separate workflow in the architecture in order to localize the changes needed as workflow changes.

## Other Architecture Patterns

There are a variety of other widely used architectural paradigms. In the following sections, we identify these paradigms and establish their relationship to the design of software architectures.

***Ad Hoc Architecture*** Ad hoc architectures are created whenever a system is integrated directly at the implementation level. These architectures contain a spontaneous mixture of communication levels and mechanisms; they are architectures without a vision. By choosing the most expedient ways to create interoperability, an ad hoc architecture can show some immediate interoperability benefits, but these benefits are short-lived and usually very dependent on particular software versions and the particular programmers who create the integration solution. These systems are very brittle; they seldom can adapt to any changes to the system, such as new software versions, new network configurations, porting to new platforms, and so forth. During the life cycle of such a system, programmers usually move on to new responsibilities, and new programmers find ad hoc systems impossible to maintain and extend.

For some organizations, such as small independent software vendors with a very stable programmer workforce, an ad hoc architecture may be

reasonable, particularly if the product is undergoing frequent revisions that impact architecture-level issues. For most other organizations, ad hoc architectures are practical only for small-scale prototypes, such as user-interface mockups.

Software architecture designs should be initiated as early as possible in prototypes that are likely to migrate into operational systems. The process of creating a good architecture for a system is very closely related to prototyping experiences and should proceed in parallel. A key goal of any operational system prototype should be the creation of a robust software architecture that can adapt to the changing requirements of the users. Ad hoc architectures do not meet these needs, except in very special organizations.

**Methodology-bound Architectures** Systems designed with rigorous software methodology can employ many architectural principles, such as component isolation and well-defined software boundaries. Occasionally, after we explain architectural principles to groups, someone responds: "We have all that, but we still cannot create a successful system." Obviously these projects are missing some ingredients.

The missing ingredients are more cultural than technical. Methodology-bound projects have a skewed value system, in which priorities in the methodology override the priorities of good architecture. For example, a commonplace methodology-driven question is: "Is that feature really object oriented?" This question may be very appropriate at an object-oriented programming level, but it has minimal relevance to software architecture. An architecture must use a wide range of concepts well beyond the scope of a single methodology. [Shaw, 93] The architecture must provide a stable solution over a decade-long system life cycle, well after the demise of today's popular methodologies. Other methodology-driven priorities include the mandatory creation of large quantities of design documentation that may be of little or no use to the developers and maintainers of the system. As the system evolves, the design documentation rapidly becomes obsolete.

In computing, we have seen many methodology trends come and go. Currently there are more than two dozen documented object-oriented methodologies. [Hutt, 94] The persistent trend is to move on to new methodology concepts.

To design good architectures, organizations must recognize that popular methodologies are transient. In fact, methodologies involve some of the most dynamic, rapidly evolving ideas in computing technology. The methodology business is also one of the most fragmented in the computing industry. There is little or no hope of standardization of methodologies; hence, there is no movement toward stability. Methodology practices will change dramatically over the course of a system life cycle. Good software architectures must provide much more stability than the methodologies that may be employed to create them.



There is probably no way to fix a methodology-bound architecture without fundamental cultural changes that allow software architects to design and implement their vision without the encumbrances of methodology-driven project culture.

***Object-Oriented Architecture*** Object orientation includes as one of its basic principles the modeling of natural real-world objects in the software system. This notion has roots in the semantic database modeling work of Dennis McLeod at the University of Southern California, where the computing object does not only represent the real-world object, "it is the object."

An important question for software architects is whether to include abstractions of domain objects at the system level. We have already said that it is not acceptable to publish unabstracted object models at the system level. Without abstractions, the architecture cannot manage system complexity effectively. There are two important considerations: (1) the stability of the domain models compared to the system life cycle and (2) whether hard-coding the domain models into the architecture provides significant system-level benefits.

Even in highly domain-dependent applications, such as simulation, the architect has a choice of whether to hard-code the domain objects into the core of the software architecture or to separate the domain-dependent hierarchies from the domain-independent hierarchies. For example, a domain-independent architecture can incorporate domain information in object method parameters as opposed to hard-coded attributes and API-level specifications.

***Client-Server Architecture*** Client-server is the precursor technology to distributed objects. Client-server technology is associated with remote procedure calls such as ONC and DCE, whereas distributed objects are associated with CORBA, which is the predominant industry standard for distributed objects. Distributed objects are peer to peer, whereas client-server is a constrained architecture with well-defined client and server subsystem roles.

Client-server technology contains some very useful concepts for the system architecture (as opposed to the software architecture), which includes hardware selection and physical allocation decisions. Client-server can be viewed as the antithesis to the migration of functionality from the mainframes to the desktop. In client-server, the focus is on creating high-performance centralized processing services. The centralized services run on high-performance systems with tightly integrated mass storage facilities. The capital resources are focused in the centralized server facilities and minimized on the desktop. When fully migrated to client-server, the desktops should be commodity terminals, such as personal computers or X Terminals. This model works because the cost of the centralized resources are shared among many users; sufficient savings usually can be obtained

from the reduced costs of the commodity desktop systems. When properly configured, the client-server systems deliver better performance at lower cost than either mainframe-based systems or PC-LAN systems.

Client-server is not concerned with interoperability. Each application functions as a vertically integrated “stovepipe” system, self-contained and providing all levels to the user from user interface to back-end mass storage. In most client-server systems, interoperability between applications is limited to desktop cut and paste—that is, exchange of text and bitmaps, not objects.

The second problem with client-server technology is that it is often vendor- and product-dependent. Since client-server products are vertically integrated, often there is no user-defined software architecture. We have heard people claim that “our architecture is <<place your favorite product here>>.” In that case, the system is vulnerable to the commercial decisions of a single supplier, and there is a lack of control of the timing of product upgrades and associated maintenance costs. Users are forced to follow the vendor’s lead or risk obsolescence. Many organizations have regretted this loss of control.

Most client-server vendors include API access to their products, so that they can be incorporated readily into user-defined software architectures. Techniques presented herein can be used with these products to create software architectures that provide vendor independence through component isolation. If a client-server product does not provide this level of API access, consider excluding it from the system. Lack of API access indicates the product has a closed, vendor-controlled architecture that could become a serious liability over the system life cycle. Even when an API is available, it should be mapped to *your own* IDL interface that *you* control.

**Message-oriented Architecture** Message-Oriented Middleware (MOM) is designed to provide message model capabilities in a distributed environment. The networking world experience has shown that messaging can be quite beneficial. Message systems provide fault tolerance. Messages can be queued, routed, or stored until a system is available for reception. The system, therefore, does not have a single point of failure. Currently MOM can be based on the RPC model, on message queueing, or on models that attempt to utilize the best aspects of both. The main benefit of this architecture is its support of both the synchronous mode of communication as well as the asynchronous mode. However, it currently is immature. Thus, developers today face multiple nonstandard APIs and questions as to how MOM will interoperate with other architectures. MOM may require more code to utilize.

## COMMENTS

The design of highly effective software architectures involves a great deal of intuitive judgment and experimentation. Rigorous methodologies that are executed mechanically can impede this creative process. The ability to create

simplifying abstractions is a key innate talent of the software architect. Few individuals practicing in the software industry have this ability—perhaps as few as one in five software designers. [Coplien, 94] The next major trend in computing methodology, “design patterns” in part resulted from recognition of this fact. The Design Pattern practitioners are focused on project-centric solutions. Their concepts are synergistic with ours, in that we both are interested in techniques for creating better software architectures. Our approach goes beyond a project-centric focus to include the creation of standard reusable architectures that can fill important standards gaps and address interoperability needs for communities of developers. We have found that many recognized authorities minimize the importance of standards. In our opinion, without organizational interface standards there is minimal reusability and no leverageable technology progress from generation to generation of software systems.

A common misconception in object-oriented systems development involves the difference between object-oriented programming and architecture-based systems integration. Many developers have practiced integration by providing system-level exposure to their object classes and methods. This practice is problematic because it multiplies the complexity of the architecture by the complexity of the internal object models of each subsystem. When a typical program comprising 100 classes and 1,000 methods is integrated with ten other similar programs, a system complexity of 1,000 classes and 10,000 methods results. These methods would be used sparsely across program boundaries in a manner that is highly dependent on specific applications. In effect, the architecture becomes highly brittle because all the subsystems become intimately interdependent upon each other’s detailed object schemas.

There is an interesting analogy here between software architecture and pop psychology. The discovery and definition of healthy boundaries is one of the fundamental principles of Gestalt psychology, the scientific basis for much of today’s pop psychology movement. Highly interdependent, brittle software systems are analogous to codependent personalities. In both cases, there is a poor definition of boundaries between independent entities. This lack of good boundary definition leads to dysfunctional behavior. In computing, this behavior is exhibited by the fact that 50 percent of software cost is attributed to system discovery (where the programmer is trying to find out how the system works). In addition, so-called codependent systems lack adaptability. The systems are unable to grow to address new requirements and challenges without great expense. Difficulty in debugging and defect correction is another important result of poorly defined system boundaries.

Object-oriented programming and architecture-based systems integration are quite different practices with very different goals. The software architecture’s role is to help control complexity and manage change in the system. Ideally, the software architecture provides a system-level abstraction that needs to contain only enough detail to provide the required sys-

tem-level interoperability. Generally designers err on the side of too much complexity with insufficient abstraction. The most important and difficult decisions are not what details to put into the software architecture but what details to leave out [Brinch Hansen, 76].

The *Rule of Three Iterations* is another important analogy for the architecture process. This analogy was explained to the author by Professor Brinch Hansen at USC; it contains some important truths. Generally, three iterations of system implementation are needed to create a mature software architecture. We can illustrate this analogy with some well-known automobile models. The first prototype resembles a Ford Pinto. It is a fairly minimal design with some obvious weaknesses. The second prototype resembles a GM Cadillac. Success with the first prototype often leads developers to try to build an overly ambitious system. This system dramatically fails to meet expectations. The third prototype resembles a Volkswagon Beetle. Due to lessons learned, it is a simple, elegant design that provides the desired functionality, but with a very economical implementation. It is hoped that the third prototype becomes the operational system, because it can be developed and maintained at a very reasonable cost.

Using this analogy as a guideline, the architecture process should continually evolve the design towards a Beetle-like system, altogether avoiding the overly ambitious Cadillac-like system. The Beetle-like system carries many benefits:

- Cost effective in declining budget times
- More versatile
- More adaptable
- More migratable
- More efficient

## Security

With the advent of open systems and networking we have seen accompanying social by-products, such as viruses, Trojan horses, unauthorized accesses, and damage by disgruntled employees. Security is an essential capability needed by the public and private sectors, in government, finance, medicine, and general business. These industries need to protect information from unauthorized access and modification, to protect privacy, to guard trade secrets, and to protect data.

Extensive research has been performed on computer security, such as secure operating systems and secure databases. In the past, not many implementations were commercially available, particularly in the challenging multilevel-security (MLS) arena. MLS security enables the interoperation of systems operating at different levels and supports multiple levels on the same system. While the number of available products and technologies that address this area is increasing, these implementations are not prevalent. MLS is still a difficult issue; without it users may not attain all of the desired benefits that distributed object management can provide while answering all of their security needs.

Because computer security historically has been viewed as a small specialty market, the commercial success of security technologies has been disappointing. Available security technologies are limited to specialized versions of platforms and products. These specialized versions are typically generations behind the state of the art in terms of cost, features, performance, robustness, and usability. In order for security to be effective it must be ubiquitous, but security is unlikely to achieve pervasive support if suppliers do not bundle it with mainstream platforms. One necessary change



is a fundamental recognition by suppliers that virtually all end users need security, not just selected narrow markets.

Some end users believe that the only available course of action is to wait until MLS security is commercially available—they ignore security or opt for physical isolation. We believe that this sort of argument should not be accepted anymore. Security must be taken into consideration, especially during system architecture design. Even though security may not be available in some of the newer emerging technologies, architects should plan on integrating and phasing it in.

Architects and developers sometimes view security as too difficult and defer it rather than build it into the architecture from the start. Often they believe that security complicates a system, imposes restrictive policies and procedures, and may impact performance. Mainframes and PCs had collocated disk packs or hard drives that were physically secure. Local area networks, remote access, and global internetworking have made security a complex challenge. Distributed systems introduce new problems, such as applications that run across several systems. When we add object-oriented technology to the equation, we find that conventional security mechanisms and architectures are woefully insufficient. The CORBA architecture attempts to provide transparent object location and activation, while the security policy may require a client to know it is communicating with a trusted object.

Security has many diverse aspects, examples of which include physical security, personnel security, information security, prevention of electromagnetic emanations and their interception, operations security, computer security, and network security (software and hardware). In this chapter we concentrate on the technology related issues.

We look further into advanced security issues that arise when using an object request broker (ORB) in a distributed system. We review several important security standards and research projects that apply to multiple levels of security architecture. In addition, in Chapter 7 we present the security support as implemented in the DISCUS framework.

### **The Need for Security**

A common misconception is that only governments are interested in security. Often software vendors or systems integrators argue that they are not concerned with security because they target the commercial world. Interestingly, while their requirements are somewhat different, security needs in the commercial world are just as prevalent as in the government.

Governments are interested mainly in protecting information (often military) from enemies. As such, they are interested both in preventing unauthorized disclosure and in maintaining data integrity. Limits are imposed on who can access information for updating and modification and also on who

can access the information for reading. The private sector is driven mainly by business decisions and their cost effectiveness. In the banking industry, protecting someone's account balance total from being read is not as important as protecting it from being modified. The private sector is therefore more interested in protecting information from being changed (i.e., restricting write access). Sometimes, although not as often as in the government or military, companies are interested in restricting *any* access to information, especially trade secrets from competitors [Chalmers, 90].

Nowadays, the needs of the government and the private sector seem to be converging. Viruses, Trojan horses, worms, and wide use of the Internet have raised the user community's awareness for security, possibly with varied emphasis. For example, the military may be concerned with data integrity in the case of data that affects nuclear weapons launches. The private sector also is concerned with the integrity of the data and wishes to protect it from unauthorized modification. This takes the form of well-formed transactions and separation of duties in the workplace. Separation of duties provides security, for example, by requiring several signatures in order to validate some operation (e.g., purchase order, release of funds, etc.). Similarly, in the military it may require several people to authorize a nuclear launch.

### Security Issues

At some level both camps are interested in authentication, auditing, monitoring, and of course good performance. Because of these requirements, care must be taken when designing a security system. Here are a few things to consider, especially when introducing security into a distributed, heterogeneous environment [Fairthorne, 94—also known as the OMG Security White Paper]:

- The security model must be independent from specific security algorithms. It must enable access to and selection of, the algorithms and support more than one security policy. This is important because different platforms and domains may have different security policies. (See section titled Security Policies on page 114.) Governments also may regulate some of the mechanisms (e.g., cryptographic algorithms), therefore, different installations of a system may need to use different algorithms. The model also should be *extensible*.
- The model must *support small systems and very large distributed systems*. This means that the mechanisms must support control and access of few users or whole groups of users. Different security policies between domains must be dealt with.
- The security policy enforced by the system *must not be bypassed* in any way. Different levels of trust have been defined by government and international criteria documents (see section titled Survey of Related

Standards on page 115). For example, if a system is being built for users who use the Trusted Computer System Evaluation Criteria (TCSEC) (see Section titled TCSEC on page 115) policies, the security model should stand an evaluation at the appropriate level of trust defined in that set of criteria.

- The White Paper notes that if an object is not responsible itself for a particular security policy, that it should be *portable* to a system that may be controlled under a different security policy. If an object is responsible for part of a policy, portability may depend on the compatibility of policies between different systems. In reality, portability of an object (even if it does not participate in any security policy) also depends on how it is implemented. An object may contain code that would introduce loopholes if it were run on a system supporting some security policies.
- The system must support object *interoperability* with other objects that may reside in different domains, that are implemented by different vendors and/or using different ORBs, and that may or may not contain security.
- Because of the wide range of security needs between different types of organizations (both within the commercial world and government), and the cost associated with various options, the system should be *flexible* to allow for a variety of choices and configurations.
- Simplicity is the key to the introduction of any system. In order for users and developers to *use* security, it should be *simple* for developers to implement it, for users to use it, and for administrators to manage it.
- Often the introduction of security can have a *performance* impact, either positive or negative. Some TCSEC evaluations have actually shown that when source code review is part of an evaluation, the improved bug detection and methods of computing actually may yield increased performance.

## BASIC SECURITY TERMINOLOGY

Security provides access to the information system for authorized users and is supposed to prevent unauthorized use of the system, its resources, data, or operation. The important objective of data protection is supported by various security services, functions, and mechanisms. To understand these better we define some of the most commonly used terms:

### Data Confidentiality

Data confidentiality is concerned with the disclosure and protection of data. Information (the data itself or the information about it) is not made available or disclosed to users (people or processes) that are not authorized to access it.

**Data Integrity**

Data integrity is concerned with the protection of the data itself from unauthorized modification. When resident in the file system or memory, or sent via communication lines, data must be protected from unauthorized alteration or destruction.

**Identification and Authentication**

The user ID identifies the particular user. In the authentication process, rights are granted to users. The authentication validates to the system that the user is indeed who he or she claims to be.

In a distributed system, identification and authentication may be required to be repeated many times. It is complicated by applications acting on behalf of other applications and the user.

**Access Control**

Access control is designed to allow controlled access to information resources. It also means that the resource should not be used in an "unauthorized manner." Discretionary Access Control (DAC) is judgment-based and depends on users granting access to objects. Often it is implemented by using Access Control Lists (ACLs). These lists of authorized users may be set up by a system administrator, for example, for system resources, or by individual users to limit (or provide) access to their own resources and files. Mandatory Access Control (MAC) is another form of access control. It enforces rules regarding which subjects may access which objects. MAC often is implemented using labels [EOSC, 92].

**Auditing**

Auditing is the process of data collection of significant security-related events to ensure that the security policy and procedures are being complied with. Data is collected in the form of system records and user activities and available for review in order to detect security breaches and unauthorized use.

**Communication Security**

Communication security is concerned mainly with data protection during transmissions. Data integrity and confidentiality may need to be enforced, and protection may need to be provided against the capture and replaying of data and messages.

**Security Administration**

Security administration has two aspects. One is concerned with access control over administrative operations. The other relates to the system policy configuration and maintenance.

**Non-Repudiation**

Non-repudiation ensures that an action originator cannot deny that it performed the action. For example, a message originator cannot deny that it sent the message and the recipients of the information also cannot deny that they received it. Digital signatures or public key encryption mechanisms are useful for implementing non-repudiation [Shaffer, 94].

**Assurance**

Assurance represents the level of trust that an element of the security architecture (software or hardware), a service, or the whole system enforces the security policy and performs its function as expected. Assurance can be gained via NSA evaluations and certification, and via testing (such as penetration testing, modelling, and simulation) [Shaffer, 94].

**SECURITY POLICIES**

A security policy comprises a set of rules and practices designed to counter some threat. Security policies allow different organizations to impose security in a manner that is consistent with each organization's mission and goals. Authorization, or identity-based policies, filter access to resources depending on the user's identity and need to know. A rule-based policy may use object labeling to determine the sensitivity of information. Often organizations that truly require a very secure environment simply impose a policy within their systems and disconnect themselves from the rest of the networked world. Such isolation may provide the required security; however, it obviously detracts from open and distributed systems. History tells us that isolation limits advancement.

Different security policies between different organizations introduce new challenges and complicates distributed environments. Guards or gateways may be needed to translate between different security policies. Furthermore, some systems may be required to support more than one policy.

Dealing with security in the architecture requires very careful farming and mining (see Chapter 4). One could design a distributed architecture that is technically sound and enables distributed communication between various applications and objects. It may be discovered later that a certain security policy prevents key information from travelling from one segment of the network or system to another.



## **SURVEY OF RELATED STANDARDS**

In this section we briefly survey some of the well-known standards and some of the important initiatives under way that may shape security criteria for the next several years. These are some of the key standards and technology that we feel are important, or could play a role, in systems integration using distributed objects.

### **ISO 7498-2 Security Architecture**

This standard defines the general security-related architecture elements that can be used to provide secure communication between open systems [ISO, 92].

The document defines the standard security services: Authentication, Access Control, Data confidentiality, Data integrity, and Non-repudiation. Security services are invoked at the proper ISO layers and in proper combinations to support certain policies and user requirements. The document also defines security mechanisms that are used to implement the basic services. These include encipherment, which is the cryptographic transformation of data to produce cyphertext; digital signature mechanisms, which are appended data that provide data integrity; access control that may involve using security labels; traffic padding, to protect against traffic analysis; routing control, to allow use of specific physically secure parts of the network; and notarization, which is the use of a third party for authorization.

The standard also defines "pervasive security mechanisms" that are defined as the Trusted functionality to establish effectiveness of other security mechanisms. As part of the Trusted Computing Base (TCB), these components are expected to be trustworthy. The document also defines security labels, event detection, audit trail, and security recovery.

The standard presents a view of the placement of different types of security as they relate to the ISO 7 layer model. It should be noted that layer 7, the application layer (which is also the layer of CORBA application integration), presents all security services.

### **TCSEC**

In 1985 the Department of Defense (DoD) published the Trusted Computer System Evaluation Criteria, also known as the Orange Book [DoD, 85]. The document defines a secure computer system and identifies six requirements that a Trusted Computing Base must satisfy in order to be considered a secure system. According to the TCSEC, a secure system is one that, through the use of security features, controls access to information such that only properly authorized individuals, or processes operating on their behalf, can have access to this information.

The TCSEC defines six requirements that a “secure” system must satisfy. Four deal with what the system must provide in order to control access to the information it manages; two deal with how assurances can be obtained so that the first four requirements may be accomplished.

- *Security policy.* The system must enforce an explicit, well-defined policy. Given subjects and objects, there must be a set of rules that the system can use to determine whether a given user can have access to a specific object.
- *Marking.* Objects must be marked with sensitivity levels (classification).
- *Identification.* The system must require individuals to identify themselves. It must control access to information based on this user id.
- *Accountability.* The system must collect audit information and protect it, so that actions can be traced back to users.
- *Assurance.* This includes hardware and software mechanisms that can be evaluated independently to provide assurance that the system is enforcing the preceding four steps.
- *Continuous protection.* Hardware and software mechanisms must be protected against unauthorized changes.

***Divisions and Classes*** TCSEC defines a set of standards for computing systems having different levels of security requirements. Security criteria is divided into four categories, A, B, C, and D. Category A represents systems with the most comprehensive security. Each division may be subdivided into classes with numeric suffixes; higher numbers represent a higher level of security. For example, B level represents higher security than C level, and B3 represents a higher level of security than B1. Most vendors today have systems with at least a C level of certification.

The Orange Book is considered somewhat outdated. It relates mainly to mainframe operating systems and is too inflexible for nonmilitary uses. TCSEC-based systems do not necessarily interoperate [Shaffer, 94].

### **Other Criteria Initiatives**

The Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria (TNI) was created because of the difficulties in extending the TCSEC as a criteria for networks. The TNI covers network partitioning into components of different ratings to support components that perform functions requiring varied degrees of assurance. Each distributed part of the network trusted computing base (NTCB) can have a separate set of objects, subjects, and security policy [EOSC, 92].

The European Community (EC) has published its own evaluation criteria known as the Information Technology Security Evaluation Criteria (ITSEC).

Canada also has its own Canadian Trusted Computer Product Evaluation Criteria (CTCPEC). These two standards and the Orange Book need to address broader functionality and assurance.

The Common Criteria (CC) is a multinational effort that began in 1993. Its main objectives are to develop open and flexible frameworks for defining new requirements for Information Technology security; to gain international recognition for evaluation results and through modernization of the process to reduce cost; to modernize security criteria to address open distributed systems and integrated computer and communications security; and to protect previous investment in security products while reducing trade barriers. While we expect that in the long term there will be international standards and a revision made to the Orange Book, currently the CC is still in development and somewhat immature. Products with new function and assurance combinations are still several years away.

## **SURVEY OF RELATED TECHNOLOGIES**

Much research has been performed in the areas of security architectures. In this section we investigate some of the more relevant technologies that may be used in distributed systems and in conjunction with object management technology. There are other technologies that are potentially relevant, but are beyond the scope of this book.

### **Kerberos**

Kerberos is a distributed authentication service developed at the Massachusetts Institute of Technology (MIT). It is a symmetric security system where a client and server share a key for encrypting and decrypting data [IETF, 93; Malamud, 92].

Kerberos simply delivers credentials, which are made of a "ticket" that contains information authenticating a user to the server and a session key required to use the ticket in the authentication transaction. Kerberos does not impose any rules on how tickets are used by clients and servers. In a normal exchange, a client requests access or "credentials" to a specific server from an Authentication Server. The Authentication Server has access to a database of client and server keys. The server sends a packet back to the client encrypted with the client's key that includes a ticket for the requested server and a session key. The client then sends the information to the server encrypted in the server key.

Tickets may be temporary and clients may need to request new tickets intermittently. The ticket is the authenticator and is encrypted by the server key so that the client can use it but not modify it. Once authentication is complete, the client and server may not require further authentication, or

they may continue to encrypt further communication. More keys, called subsession keys, also may be exchanged.

While kerberos can be used between organizations, it does not scale well to large networks [Malamud, 92; Shaffer, 94]. This is especially true in the case when some organizations use kerberos and others use other authentication mechanisms.

### **Trusted Computing**

***Trusted Computing Base*** In order for a system to function as a Multi-Level Secure (MLS) system using sensitivity labels, it must have a Trusted Computing Base [Fellows, 88]. The TCB components must be dependable in the sense that they cannot be modified by unauthorized users or processes, and they always must function correctly in performing their security-related task.

A distributed TCB must perform all of the functions that a normal TCB does, but verification is much more difficult. A TCB on a domain that encompasses many systems faces a number of problems. These problems may be straightforward (although not necessarily simple), such as different policies across different domains, and they also can be more subtle. For example, timing across components can cause inconsistencies in the security contexts and their states. This can result in states that are not ordered, especially if components are replicated across the system to provide fault tolerance.

As part of TCBs in systems that use TCSEC level B2 criteria or higher, a trusted path must exist to ensure to users and programs that they are indeed communicating with the TCB.

If a system requires complete security at each level of the ISO model (seven layers), Trusted Protocols are needed at each level. Often security is added to existing standard protocols rather than new protocols being created. As such, it is likely that security will have to be added also to the Object Management layer to guarantee end-to-end security between applications.

***Distributed TCB*** It is possible to provide a Trusted Computing Base in distributed environments by using a distributed hierarchical TCB [Fellows, 88]. Such a TCB makes use of other TCBs. If each local TCB enforces its policy, the distributed TCB is ensured that its policy is enforced. This sort of structure, though, can be complicated by various state transitions.

The consistency of states and security contexts is very important to maintaining secure systems. Even if a part of the system is down or not available, security and data must not be compromised; rather services should be denied. The states must be synchronized intermittently between the local and distributed system TCB. Marshalling of data, as is often done with the ORB, can cause replication of data. It is necessary to have vertical security between layers and horizontal security between different ORBs.

**Compartmented Mode Workstation** In 1985 the Defense Intelligence Agency (DIA) started the CMW (Compartmented Mode Workstation) program to define the security requirements for workstations that handle compartmented mode information. A compartment is a designation applied to a type of sensitive information. It indicates the special handling procedure to be used for the information and the general class of people who may have access to the information. In 1991 the CMW requirements were defined in the CMW Evaluation Criteria using the TCSEC baseline with the addition of deltas [CMW, 91]. Several vendors were contracted to produce commercial off-the-shelf workstations based on the CMW requirements, which include TCSEC B1-level features plus some B2- and B3-level features; they are referred to as a B1+ system.

By definition, CMWs restrict information sharing, flow, and exchange between processes, users, the file system, other systems, and a combination of these. To support these requirements, the window system must be able to display security labels and the window manager must control information flow between windows. A Trusted X server had to be developed to provide secure communication between X window components. The Trusted X server labels windows and other X objects as well as user input. The window manager is part of the TCB and intercepts certain events, such as cut-and-paste, between windows. If the user attempts to move information between windows with different security labels, the window manager might prompt the user to downgrade the classification of the information, if the user is appropriately authorized to do so.

This sort of small granularity event interception can cause a large performance overhead, especially in distributed object-oriented environments. It is clear that security in the ORB environment can get complicated quickly if, as in the Trusted X system, each object access, embedding, and linking may have to be checked.

**Trusted Mach B3** The developers of the trusted Mach operating system have found that using object-oriented techniques in secure systems assisted in meeting the assurance and security policy defined by the TCSEC [Gupta, 93]. The designers found that using an object-oriented language, such as C++, with the TMach kernel allowed them to implement a multilevel secure operating system utilizing the client-server model. The kernel provides process security using task IDs and isolation and intertask communications utilizing messages. The use of an object-oriented language in general provides for modularity, inheritance, and the exchange of messages. Other benefits are abstraction and data hiding. The data hiding can be provided through restricted interfaces, such as the public and private access in C++.

**CORBA-compliant Security in Synergy** Realizing that existing security models and architectures will not meet the needs of distributed systems, the



National Security Agency (NSA) embarked on a research project called Synergy to develop a portable, microkernel-based security architecture [Saydjari, 93]. Synergy plans to provide flexible support for multiple security policies, including commercial and government policies.

Synergy is an ongoing prototyping effort, and some changes to the information presented here are expected. Its architecture is planned to support networked MLS systems that use trusted gateways to access untrusted machines. The gateways label the data. At present, Synergy is concentrating on a homogeneous environment. In Synergy, separate servers exist for auditing, authentication, cryptography, and security. The servers are accessed via the microkernel, which provides a common machine-independent interface. The microkernel is based on the Mach 3.0 microkernel from Carnegie-Mellon University [Saydjari, 93].

- *Security server.* Provides access control. Policy decisions and enforcement are separate. This server controls access control only; the microkernel and other servers perform the enforcement.
- *Audit server.* Provides a centralized auditing facility that can receive messages from all parts of the system.
- *Cryptographic server.* Available to applications, this subsystem can provide access to various mechanisms and algorithms.
- *Network server.* Uses the X-kernel protocol and provides secure multi-level communication across unprotected networks.
- *Authentication server.* Provides user authentication and system authentication.
- *O/S server.* Currently built on BSD 4.3 UNIX. It should be able to support other operating systems in the future.

In Synergy, UNIX processes are single-threaded microkernel tasks. Memory objects that correspond to address space segments are mapped into the task's address space. All memory objects have security attributes, and it is up to the microkernel to enforce the security policy.

An example of how the microkernel does this is when an application uses the fork system call to fork a process. The child inherits the parent data and labels. If the child changes the label, the microkernel checks all memory buffers and resources and may prevent access to certain parent information. The microkernel uses emulators that look like standard system calls or file system calls, but performs certain additional security checks. Files are mapped onto memory objects. Memory objects have security attributes. Pipes and sockets are mapped to microkernel ports and therefore also can be checked using the microkernel mechanisms.

At the networking level, only multiple single-level connections are supported. This is because multilevel connections will require changes to networking protocols. The MLS local area network (LAN) assumes that con-

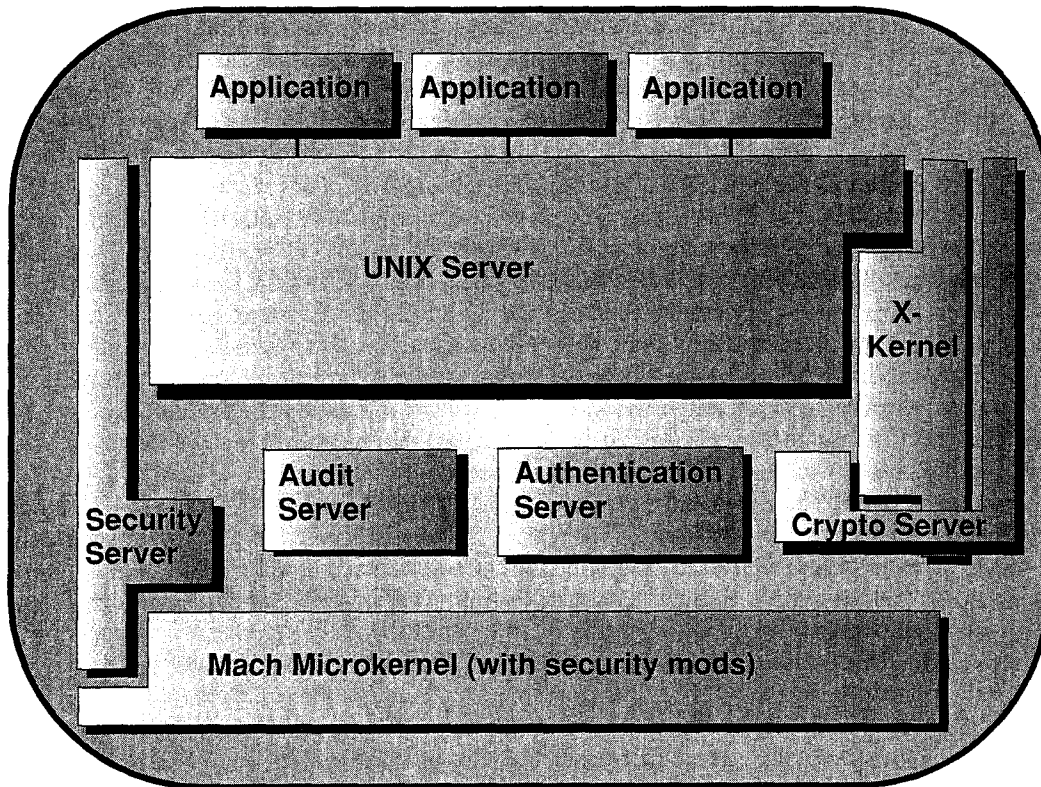


Figure 5.1. Synergy architecture.

nections are connection-oriented and performs access control only at connection establishment. This part of the system was developed as a proof of concept and does not support Connectionless data [Saydjari, 93].

X-kernel is a network protocol from the University of Arizona to support secure communications and is used in the workstation to provide network support. It supports decomposition of complex networking protocols into "microprotocols" allowing more flexibility and building blocks to build other protocols. The breakdown provides the same functionality but easier insertion of security mechanisms at each level.

The authentication server makes use of the Generic Security Service Application Program Interface (GSS-API), as do the Crypto and security servers.

### GSS-API

The Generic Security Service Application Programming Interface is intended to provide generic security services independent of the underlying mecha-

nisms or technologies [McMahon, 93]. The user of the GSS-API may be an application, or simply another protocol. In either case, the communications between entities needs to be protected. The user may be interested in authentication, data integrity, and confidentiality. The user receives a token from its local GSS-API and exchanges it with a peer application or protocol, which in turn passes it to its local GSS-API. A security context is established by the GSS-API between the two peers where secret-key or public-key cryptographic systems can be used.

The GSS-API uses structures called Credentials that allow peers to establish security contexts. Data elements called tokens also are transferred between GSS-API callers. Context-level tokens, for example, are used to establish and manage contexts. Per-message tokens are exchanged with an established context to provide security for data messages. Contexts are established using credentials, and there can be more than one context between peers.

The security mechanism type is the underlying mechanism the GSS-API uses for encrypting and decrypting that both peers support. Naming also is provided to allow for the handling of the security context as opaque octets. Channel binding allows for the binding of contexts to specific communication channels.

Here is an example of a typical GSS-API session [Wray, 93].

1. An application or process acquires credentials so that it can prove its identity to other processes. The application must not reveal the name of the user who is executing it.
2. Using the credentials, two applications can establish a joint security context. Usually the application that initiates the communication must be authenticated to the receiver; however, authentication also may be requested from the receiver to the initiator. An important feature of the GSS-API is that it allows delegation of rights to a peer and the ability to apply security services, such as confidentiality and integrity, on per-message basis. The receiver may then create more contexts on the initiator's behalf, using credentials similar to those of the initiating application. Some GSS-API calls use an opaque data structure called a token. The token must be passed from the caller to the peer, who must pass it to its local GSS-API to be decoded and extracted.
3. Some services may be invoked on a per-message basis to provide data confidentiality and integrity. The peer application may verify the data it receives, or "unseal" it.
4. At the end of the communication, the context is deleted.

The GSS-API has been very well received by many standards organizations [McMahon, 93; Wray, 93]. A list of the various organizations that have adopted the GSS-API standard follows.

- Internet Engineering Task Force Request for Comments (IETF-RFC) for base GSS-API.
- X/Open and POSIX investigating adoption of GSS-API.
- OSF DCE 1.1 to include GSS-API for non-RPC applications to access DCE services.
- ISO SC21WG6 Upper Layer Security Rapporteur's group based on input derived from GSS-API.
- European Computer Manufacturers Association Technical Committee, ECMA TC36/TG9 "Security in Open/Systems" group to develop standard to support GSS-API.
- SESAME project proposes extensions to GSS-API.

The proposed extensions for the GSS-API may be very important to the ORB environment because they have to do with the delegation of control. In the ORB environment, servers also may act as clients for other applications. For example, a map query tool may be activated on behalf of a word processor to query map databases for data. It is essential that the query map tool, working on behalf of the originating application and user, relays the correct credentials to the map servers to ensure that the user does not gain access to data he or she is not supposed to receive.

The base GSS-API does not fully support an access control security service, and the controls on delegation are not fully specified. For example, it is not possible to support selective delegation. The extensions define mechanisms that allow for the control of who can use credentials for access control or for delegation [McMahon, 93].

## SECURITY IN THE CORBA ENVIRONMENT

The requirements for security in the CORBA environment applies to objects in addition to users. Instead of controlling only user access, we must now ensure that object access also is controlled. At the same time, we must recognize that the ORB developers and OMG wish to limit the effect of security on the ORB software. Complexity can be detrimental to any standard. Simplicity, in the form of implementation hiding, minimal TCB implementation, and fewer requirements on clients and object implementations, are key to a successful security architecture in the CORBA environment.

Here is an example of how security may be used in the CORBA environment [Fairthorne, 94].

1. User logs on and is authenticated on the local system.
2. The user then activates a client application that attempts to invoke an object.
  - The user's credentials are made available to the client.
  - The client calls an object.

- The client and object need to establish mutual trust.
  - The client and object may select to have data integrity turned on.
  - The client and object establish a security context.
  - The user must be verified for the requested method execution.
  - The method invocation may be audited.
  - The object is activated.
3. The object implementation may perform its own access control against the user's credentials.
  4. If the object implementation calls another object on behalf of the client and user, it may need to use its own credentials, or the user's credentials, through delegation.

Security will be required at almost every level of the CORBA specification. For example, object creation may depend on a user's credentials and security level. Object services also will be affected. The trader service, for ex-

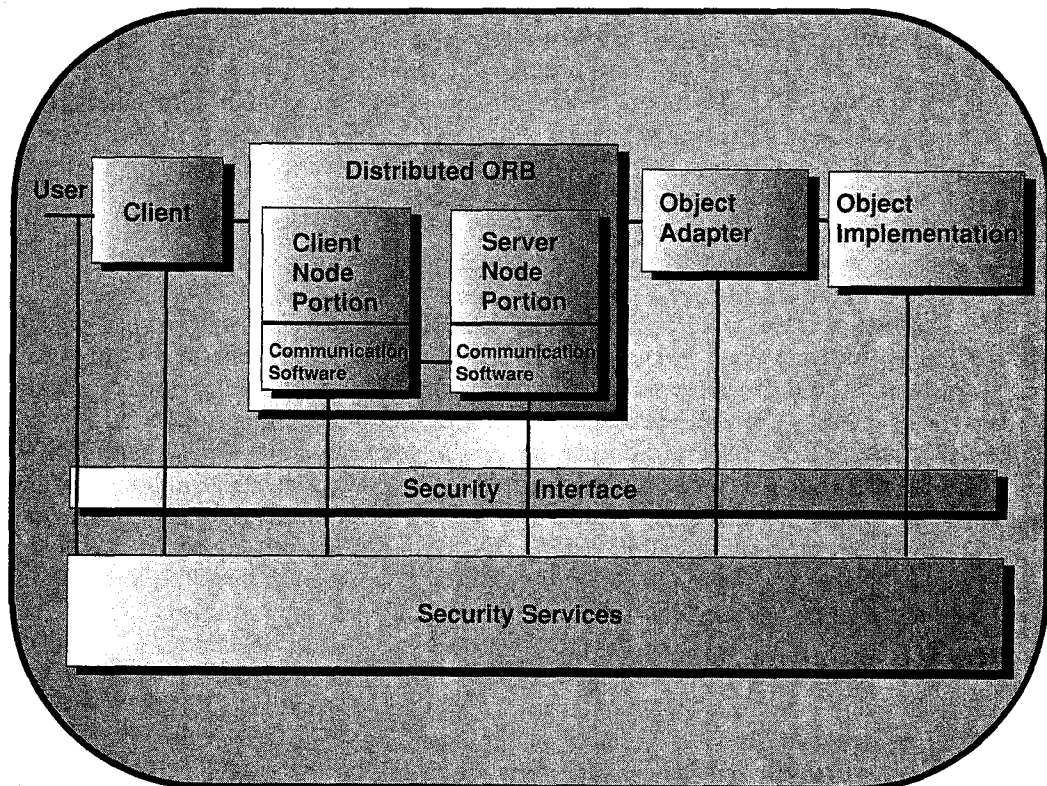


Figure 5.2. OMG white paper security architecture.

ample, may have to return to callers only information that they are allowed to “see” or access.

The GSS-API may fit the CORBA architecture very well because it provides some of the features that are required by an object-oriented system and desirable simplicity. The security mechanism used in the GSS-API may be hidden from clients. Neither the client nor the server needs to understand the security token or the mechanism in order to use it.

The placement of the calls to the security interface still needs to be determined by OMG; no doubt there will be various implementation suggestions in the responses to the request for proposals (RFPs).

The architecture provides a wide range of implementations on the server skeleton side. Access control can be performed by the ORB itself, by the Object Adapter (OA), by the object itself, or by a combination of these. Both access control and auditing should be performed by the ORB and/or OA, especially if support is required for objects that are not aware of security. This way, security can be supported across all objects, not just the ones that have implemented security themselves [Fairthorne, 94]. This is also important in isolating Trojan horse objects that can otherwise masquerade and connect to an unsecure ORB or OA.

In order to complete the secure architecture, the ORB and Object implementations should run under a secure operating system. Such a system is a key to providing a complete secure environment with file system security, auditing, and access control.

The ORB, the operating system, some communication channels, the Object Adapter, and secure mechanisms (e.g., GSS-API) may be part of the trusted computing base (TCB). The TCB also may include object services that are called by these components as well as object services that perform other security functions on behalf of other objects and clients.

The OMG White Paper identifies four possible levels of interfaces needed for the various security functions.

- *Application level.* Security parameters could be added to the Interface Definition Language (IDL) specifications to allow for the selection of security algorithms, quality of service, and so on.
- *Security aware but mechanism independent.* This is intended to be a GSS-API-like interface that the ORB can use to set up and manage security context using various security mechanisms.
- *Security services.* This interface is specific to each security mechanism and service that the software can use. It should be hidden by the security context.
- *Security service provider.* This interface should be standardized if possible because it is used to support the visible portion of the interfaces just discussed. It is used to select different mechanisms and services. This interface probably will not be standardized in response to the first security RFP [Fairthorne, 94].

The ORB introduces many complex issues. Security used to be much simpler, because we dealt only with single desktop machines or isolated mainframes. Local area networks complicated the issue, but the problem was still contained to a room, floor, building, or company. The access the Internet provides and the myriad of hardware and software protocols brought on spies, Internet worms, viruses, password cracking, and back-doors. Access to machines is easier, and holes in some communication protocols enabled security breaches and break-ins. Even today, in order to create a secure UNIX environment while still connected to the Internet, certain features must be turned off (for example tftp without -s) and other well-known bugs must be patched.

The ORB environment only adds to the complexity of the problem. In a distributed environment no longer can we simply isolate a workstation or a process. Processes may run on various machines, and a process may divide its tasks among machines. CORBA security requirements must ensure that all standard security requirements, such as auditing and authorization, are maintained and that users and objects cannot modify or have access to parts that they are not supposed to.

## COMMENTS

Security entails many interesting and controversial issues. We would like for ubiquitous security services to become available and commercially successful within our lifetimes. A fundamental problem, not unique to security, is the lack of common infrastructure and common service access mechanisms that characterize computing technology. Without commonality on these levels, security solutions are unlikely to be very effective. End-user pursuit of a fragmented set of low-level distributed computing technologies, such as Open Network Computing (ONC), Tooltalk, and Distributed Computing Environment (DCE) is just exacerbating the problem. Further fragmentation of the available distributed computing market can delay the availability of commercially successful computer security technologies indefinitely. We believe that many of the technical barriers are artificial and can be overcome by market convergence on a common infrastructure, as represented by CORBA. For example, a relatively new, immature distributed object technology, such as the Common Object Model (COM), could readily evolve toward support of OMG IDL. This logical move would unify the distributed object market for suppliers and consumers, thus enabling globally applicable security technologies to emerge.

Many key security issues are unresolved by existing standards and available technology. We enumerate some of these key issues in the list that follows. Some issues are unique to distributed objects. Others are common to both distributed objects and legacy environments. In either case, architects

and developers need to plan how to resolve these issues early in the software architecture process. Retrofit of security capabilities involves pervasive, expensive, and risky software modifications.


Security is also a very sensitive and politically charged topic for many individuals and organizations. With this caveat, here are some of the key issues (in our humble opinion):

- *Heterogeneity.* Heterogeneity of software and hardware will be pervasive in distributed object environments. In virtually all environments, there will be multiple kinds of ORBs, operating systems, integration techniques, and application packages. There are potential security issues at every boundary between these types of components. For example, consider the security implications of CORBA 2.0 interoperability. Security is an important issue at ORB-to-ORB interfaces and at other levels of the system.
- *Public metadata.* The CORBA Interface Repository and Object Trader Service make their data generally available to application software. These services may be revealing information that should be restricted by security. This is an example of general issues involving trade-offs between interoperability and security.
- *Transparent activation.* CORBA's ability to activate objects transparently leads to a new form of the Trojan horse scenario. Because the ORB can activate objects automatically, it might be tricked into transparently activating a Trojan horse object instead of the intended object. Due to CORBA's inherent transparency, this event may be difficult to detect. Since CORBA has not standardized the Implementation Repository, controlling this problem across multiple ORBs will be difficult.
- *Security and embedded objects.* Object references that are embedded in application data may lead to some interesting security issues. For example, what if there is an embedded reference to an object that should be hidden from a user? In general, how would these types of object references be detected and controlled?
- *Delegation of credentials.* Many CORBA-based application architectures will utilize multiple levels of nested invocation (such as when a client calls an implementation that calls another implementation and so forth). How and when should security credentials be delegated, and how are servers that can work on behalf of many users handled? Since it is not realistic to delegate credentials to untrusted software, delegation has important architectural implications. On one extreme it is possible that the application architecture eliminates nested invocations; this would be a severe architectural restriction. On the other extreme is the possibility that the TCB expands to encompass more of the system, such as all object services, all common facilities, and some application software; this has significant cost and schedule risk implications.



- *Security retrofitting.* Migration to a secure environment probably will involve substantial modifications to software. Applications that utilize secure facilities also require modification to support security interface protocols. In our opinion, security retrofitting is difficult if possible; security should be designed into the system from inception. Commitments to secure capabilities must be made early in the architecture process to avoid the substantial risks and costs of retrofitting. Architects should plan for evolution of the system and the security facilities as commercial technology and standards evolve.

---



THE PRACTICE  
OF SYSTEMS  
INTEGRATION



## Framework Examples

Frameworks represent reusable architectures, and there are many examples available to study. In the following sections, we examine some widely used commercial frameworks, including: Microsoft's Object Linking and Embedding (OLE), the X Consortium's Fresco, and CI Labs' OpenDoc. We also describe an important historical framework project, the Autonomous Land Vehicle, which shows the complexity of heterogeneous computing prior to CORBA.

At the time of this writing, the commercial frameworks were in initial releases, but they were being actively used by programmers. Whereas OLE represents a de facto standard from a dominant industry vendor, Fresco and OpenDoc represent future voluntary industry standards from an influential nonprofit consortium. All three frameworks address the area of in-place graphical embedding, a complex interaction between software components. OLE is defined through language-specific bindings to C++ and will support CORBA interoperability through alliance products (from DEC and Candle Corp.) and potential future Object Management Group (OMG) standards. Fresco was designed from the start to be a CORBA-compliant facility. At the time of writing, Fresco was in the X11R6 adoption process at the X Consortium. Both Fresco and OpenDoc were proposed for the Common Facilities RFP1 Compound Document specification. OpenDoc is a CORBA-compliant compound document facility from CI Labs, a consortium whose members include Apple, Novell, IBM, SunSoft, and others.

## FRESCO

Fresco is a user interface framework that supports graphics, widgets, and embedded applications [Linton, 94]. It is CORBA compliant, in the sense that all application program interfaces (APIs) are specified in OMG Interface Definition Language (IDL). The reference implementation for Fresco is written in C++ and uses a library-based object request broker (ORB). OMG IDL enables Fresco to support multiple languages (current and future bindings of OMG IDL) as well as straightforward transition of software to distributed computing.

Fresco adds many new capabilities to the X Consortium's X-Windows technologies. Its new technologies include a standard object model, distributed objects, multithreading, resolution independence, and graphical embedding. These technologies are explained in more detail later.

Fresco's standard object model is the OMG's object model. Fresco's CORBA compliance is based on its use of standard OMG IDL to define all APIs for the framework. This enables the distribution of functionality across a network; however, it also allows the use of Fresco functions within the same address space that is the default implementation. In a CORBA-based environment, the applications software is identical regardless of distribution. Most ORB products support local and remote processing, including clients and server objects in the same address space. Fresco's is implemented using a library ORB (same address space), so that the ORB overhead is comparable to function calls. The use of OMG IDL enables the later distribution of client and service functionality.

Fresco provides an OMG IDL-enabled C++ programming environment by providing a reference compiler for IDL to C++ called Ix. To demonstrate how OMG IDL can enhance the utilization of C++, Fresco provides a facility for specifying object encapsulations free from implementation. Using C++ alone to specify encapsulations often involves the subtle incorporation of implementation dependencies. For example, C++ constructor functions imply a local implementation of object creation; an object factory (local or distributed) is the appropriate facility to create location transparent objects, as specified in the COSS Life Cycle service. Fresco also eliminates the multiple redefinitions of operation prototypes typically required when programming with C++ virtual functions.

Multithreading (MT) is a capability in operating systems for supporting multiple concurrent processes within a single program. Fresco supports both MT and non-MT operating systems. Many current windowing libraries are not compatible with MT (so-called MT unsafe code); the programmer needs to stop all concurrent threads before entering MT unsafe code. MT safety is also a significant issue in the integration of legacy software of all types. Since Fresco is an MT safe library, it can be used with MT programs without modification. Fresco uses multithreading optionally, if it is available on the platform operating system. Using multithreading, Fresco can respond flex-

ibly to windows events while a separate thread concurrently performs the screen refresh.

Fresco's graphics model provides resolution independence. The primary change is an abstraction of device-dependent pixel-based coordinates. Fresco uses floating point coordinates exclusively (except if the programmer insists on access to pixel coordinates). This enables the support of multiple screens and output devices from the same application software. Thus the application software is effectively independent of output device. This feature has substantial portability benefits and also is useful for groupware applications that support heterogeneous displays. Fresco supports both 2X3 and 4X4 transformation matrices so that the framework can be used for both 2-D and 3-D applications. This is in contrast to many current graphics standards (i.e., the Graphics Kernel System [GKS] and the Portable Hierarchical Image Graphics System [PHIGS]), which are decidedly either 2-D or 3-D and require the programmer to learn two separate specifications.

In this section, embedding means combining multiple component objects into a container object to form a compound document. On the screen, most systems with embedding restrict the embedded object to a 2-D rectangular area, and the area is assumed to be opaque. Fresco's graphical embedding extends the concept by allowing objects to be arbitrary shapes (including shapes with holes); the objects may be graphically transformed (such as rotated, translated, and zoomed); and they can be visually transparent. These advanced capabilities enable many new uses of embedding, such as advanced user environments, 3-D visual simulations, and virtual reality.

The Fresco framework comprises about 40 OMG IDL interface definitions. These 40 object interfaces, an encapsulation of 150 underlying Fresco implementation classes, form a very flat hierarchy with most interfaces inheriting from a few common base classes.

The Fresco framework has four key object types: Viewer, Glyph, Style, and Inset. Viewer objects provide a user interface to data. They control input handling, focus management, menus, and color palettes. Glyph objects are graphical objects. Glyphs also control geometry management and screen updates. The style objects provide resource attributes, which can be modified at runtime. Inset objects store the data that is displayed by other objects. Conceptually, the Fresco framework is not unlike the well-known Model-View-Controller (MVC) framework from Smalltalk [Goldberg, 83]. Inset plays the role of the MVC Model, and the Fresco Viewer and Glyph play the roles of MVC View and Controller with a different partitioning of responsibilities. Fresco supports more flexible and extensible relationships between multiple instances of each object type than found in MVC.

The following program provides some OMG IDL specifications from the Fresco sample release in May 1994, including the interfaces for Glyph and Viewer. At the time of writing, sufficient documentation to describe these interfaces in detail was not available; however, the sample programs in the

release do provide several examples of how these interfaces are used. We look forward to the X Consortium and OMG standardization of *Fresco* and subsequent commercialization as *X11R6* becomes the dominant release of *X Windows*.

```

interface Glyph : FrescoObject {
    struct Requirement {
        boolean defined;
        Coord natural, maximum, minimum;
        Alignment alignment; };
    struct Requisition {
        Requirement x, y, z;
        boolean preserve_aspect; };
    struct AllocationInfo {
        Region allocation;
        TransformObj transform;
        DamageObj damage;
    };
    typedef sequence<AllocationInfo> AllocationInfoList;
    attribute StyleObj style;
    TransformObj transform();
    void request(out Glyph::Requisition r);
    void extension(
        in Glyph::AllocationInfo a, in Region r);
    void shape(in Region r);
    void traverse(in GlyphTraversal t);
    void draw(in GlyphTraversal t);
    attribute Glyph body;
    GlyphOffset append(in Glyph g);
    GlyphOffset prepend(in Glyph g);
    Tag add_parent(in GlyphOffset parent_offset);
    void remove_parent(in Tag add_tag);
    void visit_children(in GlyphVisitor v);
    void visit_children_reversed(in GlyphVisitor v);
    void visit_parents(in GlyphVisitor v);
    // screen update
    void allocations(out Glyph::AllocationInfoList a);
    void need_redraw();
    void need_redraw_region(in Region r);
    void need_resize();
};

interface Viewer: Glyph {
    Viewer parent_viewer();
    Viewer next_viewer();
    Viewer prev_viewer();
    void insert_next_viewer(in Viewer v);
    void insert_prev_viewer(in Viewer v);
};

```

```

void insert_first_viewer(in Viewer v);
void insert_last_viewer(in Viewer v);
void link_next(in Viewer v);
void link_prev(in Viewer v);
void remove();
void insertion(in Viewer v);
void removal(in Viewer v);
Focus request_focus(
    in Viewer v, in boolean temporary);
boolean receive_focus(
    in Viewer v, in boolean primary);
void lose_focus(in boolean temporary);
boolean first_focus();
boolean last_focus();
boolean next_focus();
boolean prev_focus();
boolean handle(in GlyphTraversal t, in Event e);
void close();
};

```

## OBJECT LINKING AND EMBEDDING (OLE2)

Object Linking and Embedding, Version2 (OLE2), is an object-based framework for desktop application interoperability. It is one of the two foundational APIs (WIN32 is the other) that are supported by Microsoft Windows 3.1, Windows NT, and future systems from Microsoft, such as Windows 95 and Cairo. Current Windows 3.1 supports the legacy DOS and Windows APIs as well as OLE2 and WIN32 APIs. Windows 95 is a transitional step in Microsoft's strategy to migrate the software vendors and platform base toward Cairo, which is a follow-on to Windows NT.

A key feature of the OLE2 framework is its support for application embedding. This provides seamless application integration at the user interface. Similar to Fresco, OLE2 supports application embedding, although in Microsoft's terminology it is called in-place activation. In-place activation allows a container application to display component objects from multiple applications. When the user selects a component object, the container's menus change to allow the user to edit the object using the component application's operations. In practice, Microsoft has found intuitive user acceptance for this form of interoperability.

Microsoft is supporting OLE2 on Windows, Windows NT, and Apple Macintosh platforms, which are the target platforms for the Microsoft Office products. Since OLE2 has many implementation restrictions that limit it to a single machine, distributed extensions will support only a subset of the OLE2 framework. In particular, the DEC/Microsoft Common Object Model will distribute only persistence, monikers (naming), and data trans-



fers, but not compound documents. The Common Object Model is a future distributed version of the nondistributed OLE2 Component Object Model, to be discussed.

### Technical Description of OLE2

OLE2 is a complex collection of related technologies. It is organized so that developers can implement application support for OLE2 incrementally. Figure 6.1 provides an overview of the technologies of OLE2. The communication infrastructure of OLE2 is the Component Object Model. It defines the basic interface mechanisms for invoking OLE2 objects. Compound files is an object persistence facility that replace normal file-based storage with a more sophisticated facility for storing the data from multiple applications within a single file. This is a key enabler of object embedding, the capability to store multiple objects within a container object. Embedding supports OLE2's in-place activation, which is the capability to edit objects

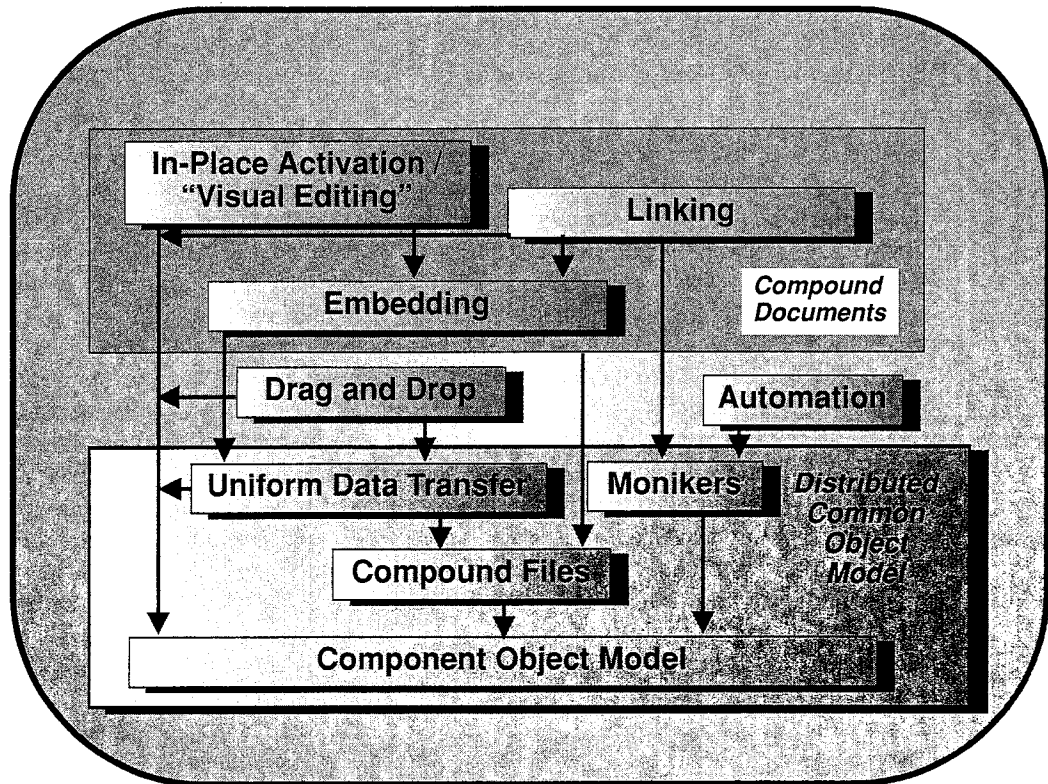
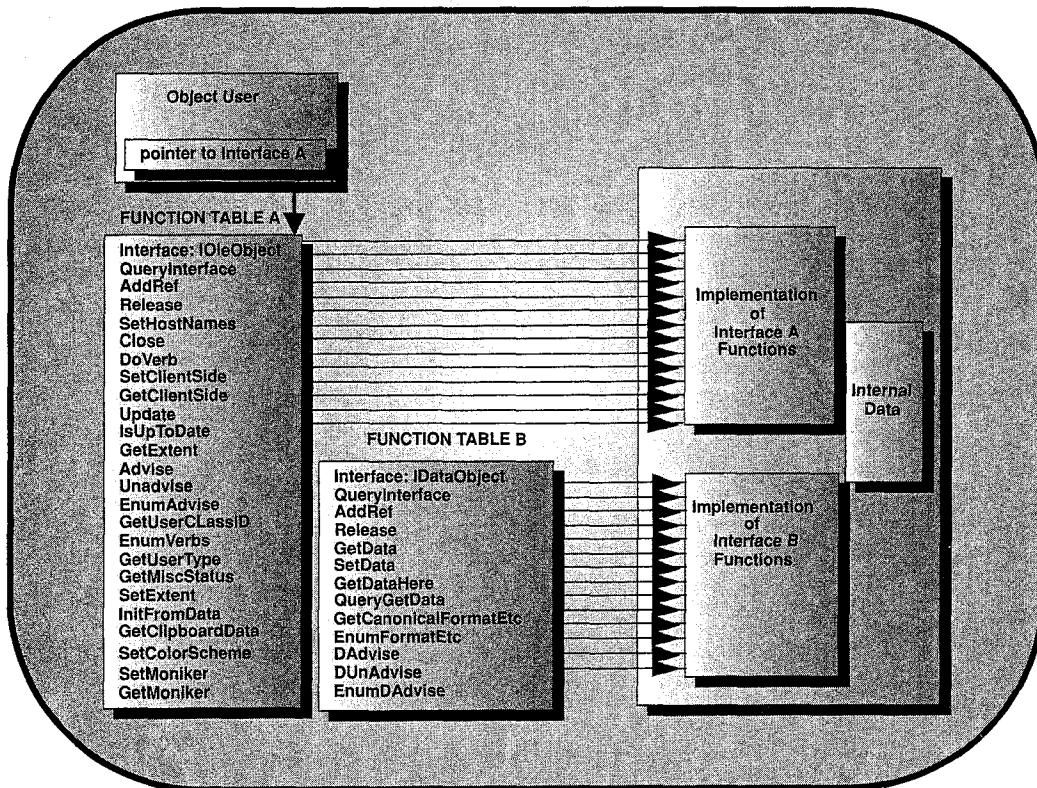


Figure 6.1. OLE/COM technology overview. Reproduced by permission of Microsoft Press. All rights reserved.



**Figure 6.2.** Component object model. Reproduced by permission of Microsoft Press. All rights reserved.

within a container object without creating a separate application window. Uniform data transfer is an upgrade to the Windows clipboard facility that adds OLE2 data objects to the clipboard. OLE2 drag and drop is a finer-grain extension of Windows file-based drag and drop. In OLE2, subsets of documents may be dragged and dropped between similarly enabled OLE2 applications. Whereas OLE2 embedding results in a separate copy of the source data, OLE2 linking supports the display of common data in multiple documents with updates. The moniker facility provides a naming capability. The implementation of monikers is server-specific; names make sense only to the application that creates them. Monikers also support linking using file pathnames. OLE2 Automation is the capability to control applications electronically through a dispatch function that is similar to a nondistributed version of CORBA's Dynamic Invocation Interface (DII).

OLE2 Component Objects are invoked through function tables. Object invocations are equivalent to calls to dynamically linked libraries (DLLs). Each function table contains three or more function pointers, since all ob-

jects must support three base functions from the interface IUnknown. OLE2 predefines more than 60 interfaces, each containing a number of APIs. User-defined interfaces are possible but require the user to implement custom marshalling code, a process that is not documented comprehensively. For most practical purposes, OLE2 is intended to be used with the predefined interfaces provided by Microsoft.

Each component object implements one or more OLE2 interfaces. A client application with an interface pointer can request other interfaces through the function QueryInterface, which is present in all interfaces. An object can grant or deny access to its interfaces based on its response to QueryInterface. The notion of having an object identifier as a complete reference to an object is purposefully not supported in OLE2. This enables OLE2 objects to grant access to their component interfaces selectively.

Uniform Data Transfer (UDT) replaces several data interchange mechanisms present in DOS and Windows, such as file drag and drop, dynamic data interchange (DDE), and OLE Version 1. UDT reuses the existing Windows clipboard with a new protocol based upon OLE2 Data Objects. Data Objects support the interface IDataObject, which provides a new facility for transfer of formatted data. Data objects may be transferred through the clipboard or through OLE2 drag and drop.

The OLE2 clipboard framework operates as follows (Figure 6.3). The source for the data creates a data object and posts its IDataObject interface pointer to the clipboard using the DLL function OleSetClipboard. The data source can post several alternative data formats to the clipboard to increase the probability of transferring a common format between the source and consumer. Once the data object is on the Windows clipboard, its implementation is provided by the OLE2 DLL and delegated back to the original object. The consumer can obtain the DLL's Data Object interface pointer using the function OleGetClipboard; then the consumer can retrieve data using Data Object functions such as EnumFormatEtc (to obtain a list of potential formats), QueryData (to determine if request for a specific format would succeed), and GetData (to retrieve the data) [Microsoft, 94a].

Secondary storage mechanisms in OLE2 are defined as a specification called structured storage. OLE2 provides an implementation of this specification called compound files. Compound files enable the storage and partitioning of complex data from multiple embedded objects into a common file. Compound files replace the need for ad hoc approaches such as multiple separate files or complex storage schemes. An example of a compound file is shown in Figure 6.4 on page 140. Compound files are created and managed by OLE2 container objects. Container objects allocate storage in the compound file for embedded objects that perform their own input/output functions using OLE2 persistence APIs. When an embedded object is activated, it is passed a pointer to its compound file storage. Compound files comprises 9 interfaces with more than 70 API functions to support these capabilities [Microsoft, 94a].

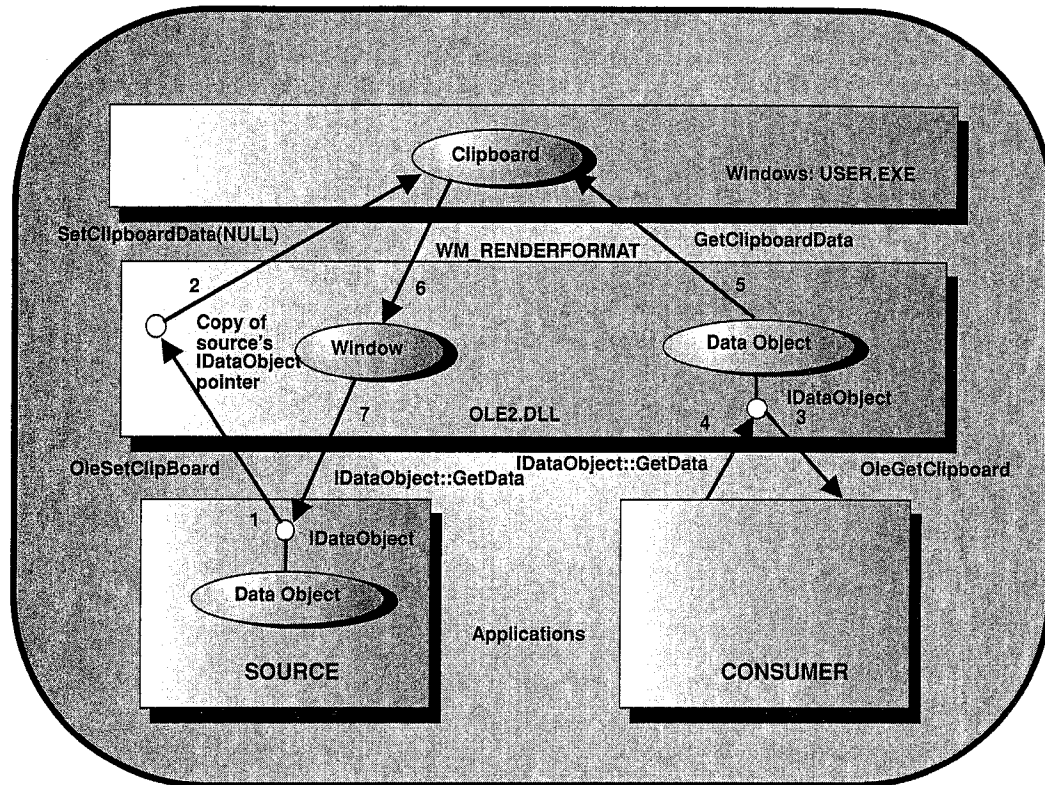


Figure 6.3. Clipboard framework. Reproduced by permission of Microsoft Press. All rights reserved.

### Comparison of OLE/COM and CORBA

In this section, we compare and contrast the features of OLE/Common Object Model with CORBA. OLE/COM is the next generation of OLE2 with some distributed capabilities for persistence, naming, and data transfer. In this discussion, we refer to OLE2 as needed, because it represents the currently available technology. Our perspective is in terms of a software architect evaluating the two mechanisms as the basis for an end-user system.

Object-oriented system developers must choose between OLE/COM and the Object Management Group's CORBA. This choice is an important one, because these two mechanisms comprise the future technology direction of the volume platform vendor and the balance of the computing industry, represented by the 500+ members of OMG.

Both OLE/COM and CORBA provide general capabilities for application integration. CORBA is a general-purpose communications infrastructure for object-oriented software. It is an industry standard supported by multiple platform vendors and independent software vendors. OLE/COM provides an

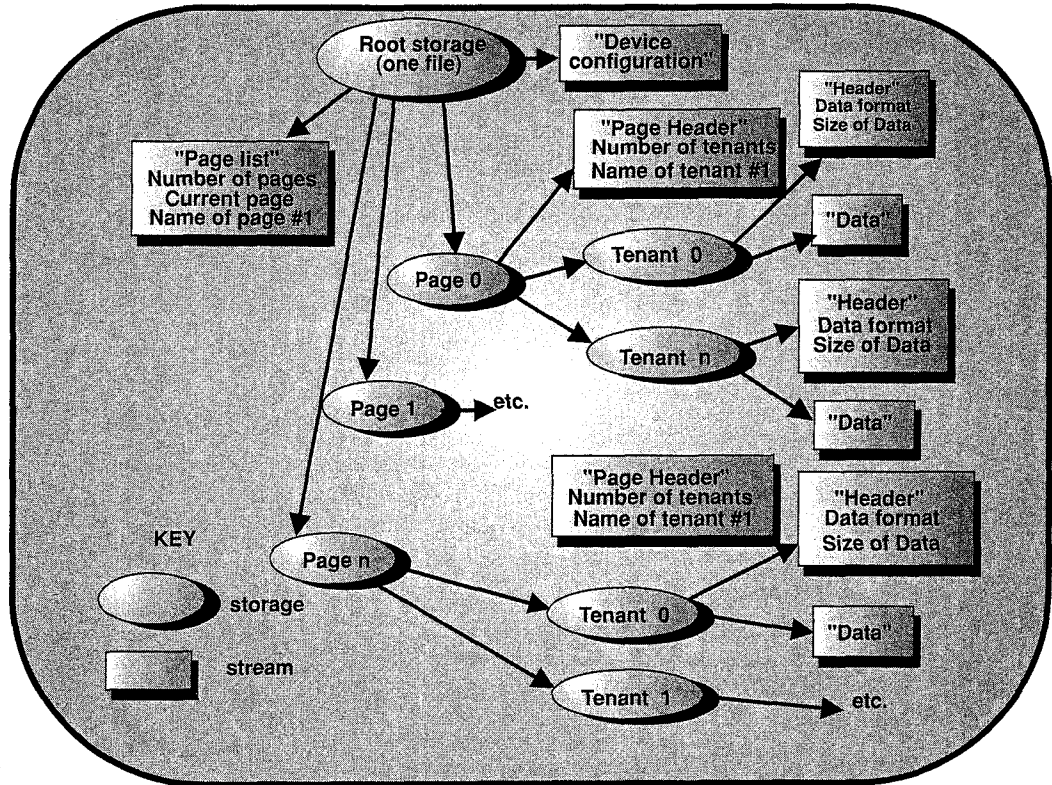


Figure 6.4. Structured storage. Reproduced by permission of Microsoft Press. All rights reserved.

object-based software infrastructure, but it is better known for its compound document framework.

This discussion examines the relative trade-offs between OLE/COM and CORBA. Suppose we are reengineering an end-user application system, a commonplace activity in today's computing environment. The end-user system includes a combination of custom software and commercial off-the-shelf software. A mixture of computing platforms is present, since no one type of computer meets all needs. The goal of the reengineering activity is to provide data sharing and interchange between all units of the organization to foster increased productivity and flexible redefinition of end-user roles. The resulting system should be adaptable to new requirements during its life cycle and should minimize costs for development, operations, and maintenance.

For the system developer, some key criteria defining the choice between the OLE/COM and CORBA include: (1) the generality of the mechanism and its adaptability to the integration problem; (2) the mechanism's support for extensibility; (3) the cost implications of each mechanism; and the (4) stability of the mechanism's technology over the system life cycle.

**Generality of OLE/COM vs. CORBA** A first consideration in the comparison of CORBA and OLE/COM is the generality of the mechanism and its adaptability to the software integration problem. Some key factors include support for distributed processing, multiple platforms, and integration with other communication mechanisms.

*Distributed Processing* Distributed processing is a key requirement for today's increasingly networked systems. A challenging activity for programmers, distributed processing is costly due to its inherent difficulty. When software is communicating across a distributed system, there are many possibilities for errors, failures, and unpredictable events. The distributed software must be configured correctly, in the appropriate running state, and issues such as the machine-level data representation between the machines and the network must be resolved for software to execute as desired. CORBA is a distributed processing standard defined to simplify these issues for the programmer. It automatically handles data conversions, server activation, and reliable handling of errors. The major object request broker products all support these distributed processing advantages through the CORBA standard.

In contrast, current OLE2 technology does not support distributed computing. The current OLE2 product and supporting frameworks are designed for single-user, single-machine applications. Distributed computing in the OLE/COM environment will be supported by a new mechanism based on proprietary extensions to Open Software Foundation Distributed Computing Environment (OSF DCE).

*Heterogeneous Platforms* Integration across multiple operating system platforms is a key requirement in many end-user environments. A mixture of different platforms is an unavoidable fact of life in most organizations, because many jobs require the specialized functions of several computing systems. The ability to interchange data between different platforms would offer a significant productivity boost in these environments.

Current CORBA ORB products support software integration across multiple platforms. Using today's ORB products, each product bridges multiple platforms, providing a consistent set of application software interfaces on any platform. In the future, CORBA products will be bundled with many platforms, and these ORBs will interoperate with other CORBA implementations. The developer will have the choice of using a single ORB across multiple platforms or multiple interoperating ORBs. Some products, such as DEC ObjectBroker, IBM System Object Model (SOM), and IONA's Orbix, are targeting the current and future multiplatform markets. While products such as Hewlett-Packard ORB+ and SunSoft DOE are focused on a single platform, these vendors have recruited allies to provide mutual ORB interoperability solutions.



OLE2 offers platforms for Windows and Macintosh. The Windows platform is supported more because it is described in the published OLE2 documentation [Brockschmidt, 94; Microsoft, 94a; Microsoft, 94b]. Coincidentally, these are the platforms where Microsoft's Office products are deployed. DEC's alliance agreement with Microsoft for the OLE2/CORBA Common Object Model gives DEC the charter to cover other platforms, and other vendors are planning to supply CORBA products for this niche. Available OLE2 technologies do not address multiplatform interoperability since it does not support distributed communication between Windows and Macintosh. OLE2 is not a multiplatform technology, but it may migrate that direction through third-party CORBA support, such as through OpenDoc.

*Multiple Communication Mechanisms* Integration with multiple communication mechanisms is an important capability to support software integration. Because so many mechanisms (sockets, Open Network Computing [ONC], DCE, TOOLTALK, etc.) are available, any given assortment of legacy and commercial software will utilize a variety of them. A key role of object-oriented software architecture is to encapsulate the differences between these mechanisms and provide a consistent system integration solution. The technology chosen to resolve these issues must offer support for encapsulation that hides underlying implementation differences.

The CORBA specification is explicitly defined to be independent of underlying communication mechanism. Any given communication mechanism can be encapsulated behind an OMG IDL application program interface, and this encapsulation is devoid of implementation detail. ORB developers and CORBA users frequently apply this concept, encapsulating many levels of communication mechanisms behind OMG IDL APIs.

OLE/COM does not have any explicit support for the integration of multiple communication mechanisms. Its underlying communications mechanisms are provided by Microsoft and are not suitable for replacement by developers or multiple suppliers.

*Example Object Identification in CORBA and OLE/COM* Object identification is one of the fundamental concepts in object orientation. CORBA provides a unique object identifier for each object instance. Its object identifiers (OIDs) are opaque to the programmers. CORBA makes issues such as differences in location, language, and operating systems transparent to clients. The object identification mechanism is supported by the COSS naming service, and the trader object service that provides white pages and yellow pages directory services.

OLE/COM does not have a comparable concept of object identity. OLE/COM discourages the notion that anyone can have a pointer to an entire object [Brockschmidt, 94]. Instead clients can have transient pointers to particular interfaces of an object. OLE/COM also has an object naming facility

called a moniker. File monikers store absolute and relative file pathnames to provide two ways to find a file. The absolute pathname is tried first; then the relative pathname is sometimes successful if an entire directory structure has moved. This is an incremental improvement upon OLE1, which lost its file links whenever files were renamed and moved. Item monikers are used by an object to define its own namespace. OLE2's lack of object identifiers is an impediment to distributed processing, in which location-dependent names do not translate across machine boundaries.

***Extensibility of OLE/COM vs. CORBA*** Developers of object-oriented software architectures require mechanisms that support user-defined objects. The extensibility of the mechanisms to support new standards and alternative implementations of predefined specifications is also important for the tailoring of the mechanism to meet application needs. CORBA and OLE/COM differ dramatically with respect to these criteria.

***User-Defined Interfaces*** Support for user-defined interfaces is fundamental to the implementation of application software architectures. User-defined interfaces are also essential to object orientation, which is based on a concept of domain objects that represent real-world entities or concepts. User definition of these objects is an essential requirement, since these domain-specific objects cannot be encapsulated adequately with vendor-predefined data types and interfaces. Features to be supported in the user definitions include all details of object encapsulations. For example, encapsulations should include declarations of object types, specifications of method signatures, definitions of parameter data types, declarations of attributes, and relationships between object types.

CORBA supports user-defined interfaces through the OMG's Interface Definition Language. OMG IDL is a comprehensive specification language supporting all the just-mentioned features of object encapsulations. OMG IDL interfaces are independent of programming language, so that one specification applies to multiple language bindings. OMG IDL specifications can be compiled into language-specific header files and stub functions. These files support static compile-time checked-method invocations in a form that is natural to the programming language. Dynamic binding to objects is implemented by the ORB mechanisms, and there exists a dynamic invocation interface to support runtime-constructed invocations. Early ORB products (1991–1992 timeframe) were based entirely on the DII without support for static invocation stubs. In practice, the OMG IDL-generated static stubs are a much preferable interface approach. Static stubs allow a seamless programming language interface to objects, which supports encapsulation through compile time parameter checking, efficiency, and ease of use.

OLE2 supports user-defined interfaces through two approaches. Users can write their own marshalling code to support new OLE2 interfaces [Brock-



schmidt, 94]. Writing marshalling code is analogous to rewriting the internals of an object request broker; this process is not comprehensively documented for OLE2 [Microsoft, 94a].

The second approach is supported through OLE2's interface IDispatch [Microsoft, 94b]. The IDispatch interface is similar to CORBA's DII as utilized in early CORBA products. The user defines an object's interface in the Object Description Language. ODL is compiled into a type library description for use at runtime. In order to send the message, user code must assemble a dynamic parameter structure using OLE2 APIs and call the IDispatch function invoke. A parameter structure is returned from the invocation with any exception information. IDispatch is the central interface of OLE2's automation facilities, in which OLE2 provides mechanisms for multiapplication controls.

In the future, COM will support user-defined interfaces through a proprietary extension to OSF DCE's interface definition language, called Microsoft IDL. This mechanism supports language-level and binary-level APIs that are unique to Microsoft's OLE/COM implementation.

*Extensibility of Predefined Services* In addition to the user-defined objects, any predefined interfaces must be adaptable to the needs of applications. For example, an event notification facility should be general purpose enough support application events in addition to the predefined events provided by the vendors.

CORBA-related standards for predefined services includes the Common Object Services Specifications (COSS) and the Common Facilities Specifications. The COSS services are fundamental object service specifications that are intended to be globally applicable solutions that can be specialized for particular domains. COSS and the Common Facilities Specifications are the consensus standards of the multivendor supported industry, and many adopted specifications are already available. For example, the COSS event notification service is a generic reusable event service that provides a comprehensive selection of event supplier and consumer integration approaches [OMG, 94b]. The ORB vendors will supply the implementations of COSS services, and the users can readily reimplement the services to tailor the implementation behavior for their own purposes. The OMG standards do not predefine the domain content of these services, so users can define them as needed.

In its 60+ interfaces and nearly 400 APIs, OLE2's service definitions are specialized to particular OLE2 framework needs, such as uniform data transfer or structured storage. These framework interfaces support specific forms of interoperability that are predetermined by OLE2. It is clear from their design that the interfaces are not intended to be global solutions that are tailorable to application requirements. For example, the interface IAdviceSink provides the OLE event notification service. IAdviceSink is a

special-purpose interface for OLE2 container objects to receive notifications of specific types of events [Microsoft, 94a]. It has method definitions, including `OnDataChange`, `OnViewChange`, `OnRename`, `OnSave`, and `OnClose`. The method signatures predefine the specific change indicated in the event, such as a change to a new clipboard data format in method `OnDataChange`. These particular events would not support arbitrary uses of an event service; the OLE2 service would not be extensible to support application-specific needs.

***Cost Implications of OLE/COM vs. CORBA*** OLE/COM and CORBA are new technologies that may have substantial impacts on end-user systems; it is important to examine the potential impacts of these technologies on life cycle cost. Some key cost factors include: savings from use of commercial off-the-shelf products, costs related to the complexity of the mechanisms, support for reuse, and support for portability. The portability issue can impact cost when a system must be deployed and maintained on multiple platforms. An important related issue is the need for multiple system builds to support different configurations, programming languages, memory models, and so forth.

***Commercial Off-the-Shelf vs. Custom*** A key argument for adopting technologies like OLE/COM is the potential for no-cost integration with shrink-wrap commercial applications. Life cycle experiences indicate that cost advantages of using commercial applications are not always clear. It is well known that operations and maintenance consumes 70 percent of life cycle costs. Hidden maintenance costs are associated with the lack of control end users experience when choosing off-the-shelf over custom software. Commercial vendors control the features in the product; they also control when new versions are released and the APIs that any end-user software may depend on for interoperability. When many off-the-shelf products are involved, reintegration of upgraded software is a continual maintenance activity. In some cases, maintenance agreements and licenses can lapse, such that repurchasing the software becomes necessary. Reintegration costs can be substantial when a product upgrade of two or more versions is required after repurchase.

The potential for shrinkwrap off-the-shelf integration through CORBA can be supported in several ways: directly through OMG standards, indirectly through gateways, and directly through end user-sponsored specifications.

Future CORBA-based standards based on technologies such as OpenDoc can provide direct shrinkwrap integration. Because OpenDoc is defined and controlled by multiple vendors, suppliers and consumers view these specifications as a much lower risk than a single-vendor, competitor-controlled technology like OLE/COM.

Integration with non-CORBA mechanisms (OLE2, ToolTalk, etc.) can be supported through gateways to end-user software architectures. In this ap-

proach, no-cost shrinkwrap integration is supported through the gateway, and the software architecture isolates the application code from direct dependence on the external mechanism. If the external mechanism changes, operating and maintenance costs are limited to the gateway code. Multiple external mechanisms (OLE2 and OpenDoc) could be supported through this approach. As complete industry convergence is unlikely at the level of compound document architectures, this is likely to be an important approach.

The third approach for CORBA/commercial off-the-shelf integration using CORBA is through end user-sponsored standards. End users pursuing system development frequently produce quality interface specifications. In many cases, these specifications are years ahead of comparable standards activities in the commercial market. Through coordination with others, groups of end users can form new markets by adopting common specifications. Involving vendors in an open specification process is also effective in achieving commercial buy-in. Sufficient market development can prepare a standard for formal adoption by standards groups.

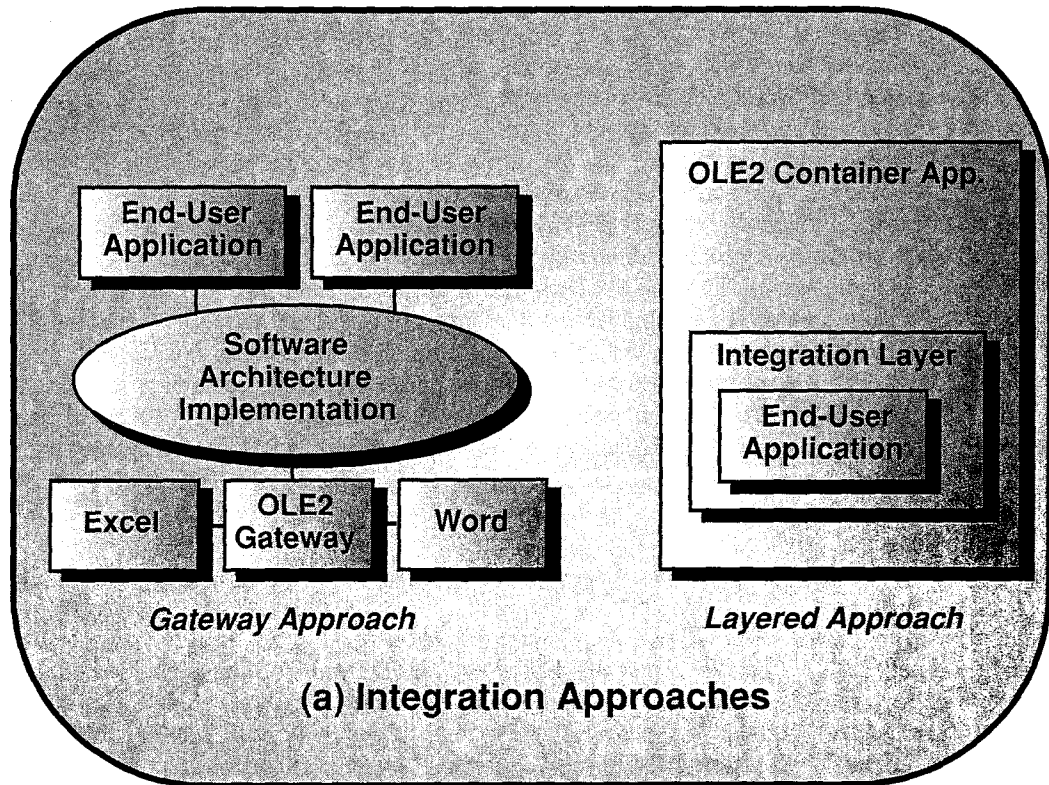


Figure 6.5. Gateway integration of software architecture and OLE2.

*Complexity of Mechanism* An important cost factor involves the integration of custom software with OLE/COM and CORBA. The complexity of the APIs, the documentation, and the available training are indicators of how costly each mechanism will be to learn and utilize.

OLE2 is one of the most complex technologies ever to be released by Microsoft [Brockschmidt, 94]. Brockschmidt, author of *Inside OLE2*, characterizes OLE2 as being significantly more complex than the Windows operating system [Brockschmidt, 94]. The documentation for OLE2 consists of the *Inside OLE2* developer's guide (925 pages), the API Reference (650 pages), and the OLE Automation Reference (400 pages) [Brockschmidt, 94; Microsoft, 94a; Microsoft, 94b]. Even at 925 pages, *Inside OLE2* does not cover significant parts of OLE2, such as OLE Automation. To use OLE2, a detailed understanding of the operating system's API (WIN32) is also required [Brockschmidt, 94]. Developer's experiences indicate that OLE2 requires a very steep learning curve. There are some development tools, such as visual basic, which simplify OLE2 integration.

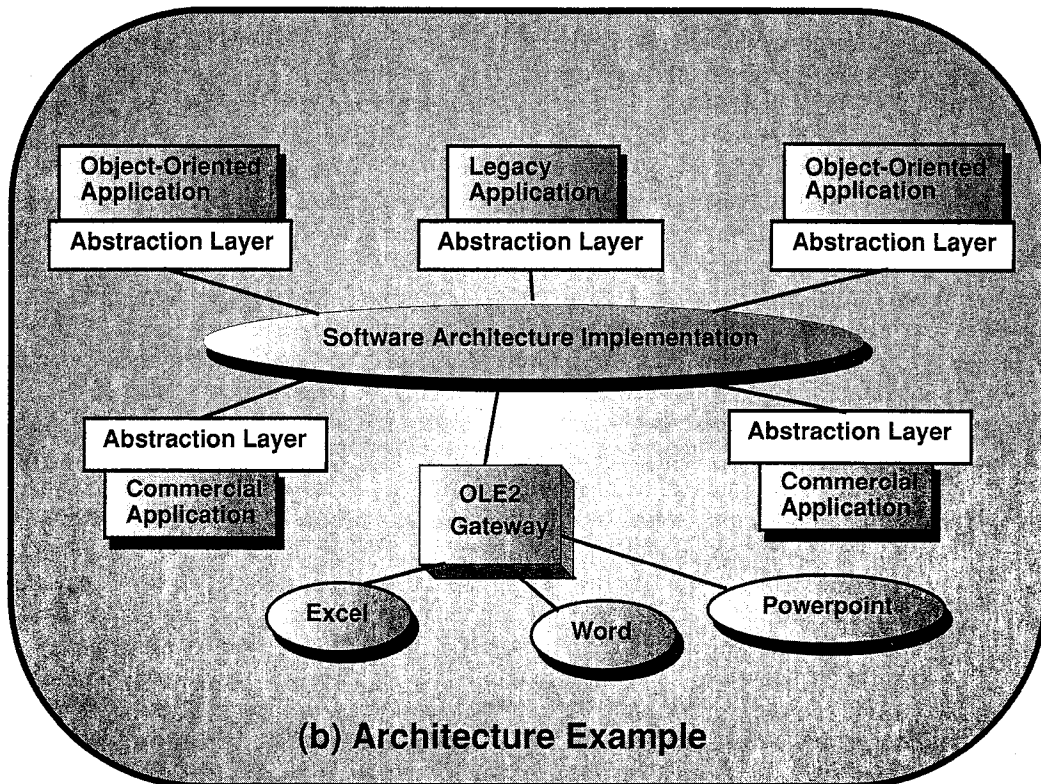


Figure 6.5. (continued)

In contrast, the CORBA specification is defined in 178 pages. Most CORBA products provide their complete documentation in another 200 to 300 pages. Training is widely available; virtually every CORBA vendor has a one-week hands-on training course, and at least four independent training companies offer general training on CORBA. With appropriate training, user experiences are very positive: CORBA is easy to learn and apply. Quite a few independent software vendors, including Bluestone, Forte Systems, Tivoli, Oberon, and Netlinks, have released CORBA-based products.

One interesting aspect of CORBA is that the system developer can control the complexity of the APIs in the application system. Using OMG IDL, developers can define an application-specific software architecture. The architecture can be tailored for the appropriate integration cost. OMG IDL is a good tool for defining software architectures and delivering the resulting APIs to large software projects that involve multiple development organizations. This concept has been applied with success in the DISCUS Framework. (See Chapter 7.)

*Support for Reuse* Reusability of code and of design is an important source of cost savings. Inheritance is the principal object-oriented mechanism for code reuse and design specialization.

CORBA directly provides inheritance through subtyping in OMG IDL. This is inheritance at the interface level only, which enables design reuse. A number of CORBA products (IBM SOM and others) are also supporting implementation inheritance, which is a direct form of code reuse. With implementation inheritance, it is easy to reuse existing objects, redefine parts of objects, and specialize their behaviors. IBM SOM uses this in an effective way to provide a reusable replication framework to support development of group applications. (See Appendix.)

OMG IDL also has significant potential for design reuse. It is a specification language that is devoid of implementation information and can be applied to multiple programming languages, operating systems, and distribution schemes. Some have called it a standard for defining other standards, because it is a general-purpose notation that provides multiple implementation bindings from a single specification. OMG IDL allows the architecture to be product independent.

OLE/COM does not support inheritance but another related construct called delegation. In delegation, to reuse code, the programmer provides method implementation that contains a call to the reused code supplied by another object. This is a less controlled form of reuse than that provided by object-oriented systems. It is really programmer-supported reuse, instead of support for reuse by OLE/COM. Microsoft believes that there is minimal need for inheritance in its third-party software market.

*Portability and Multiple Builds* Portability is an important cost issue because many applications need to support multiple target platforms, such as

different variations of UNIX, DOS/Windows, Macintosh, and other operating systems. A related issue is the need for multiple builds of software to support interoperability with other languages, memory models, and other factors.

One of the key advantages of CORBA is that the APIs are consistent between all platforms. When standardized, the OMG IDL language bindings are consistent for that language across all platforms. In general, CORBA will not be a cause of platform dependency; other factors such as the operating system APIs and windowing system APIs are potential sources of portability problems. These are addressed through other standards, such as POSIX and MOTIF, with significant success. CORBA eliminates the need for multiple software builds when client and server are implemented in different languages. Products such as IBM SOM also isolate clients and servers from differences in memory models and other implementation factors.

OLE2 is best supported on Windows platforms. It is available, but not as well documented, on the Macintosh. Third-party vendors (such as DEC) will support a subset of distributed COM functionality on other platforms. The OLE2 APIs as supplied by Microsoft are a significant source of platform dependency now and in the future. Since OLE2 defines explicit bindings to Microsoft C++, OLE2 is also language dependent and will require significant developer effort to provide alternative language bindings and code reuse between languages.

***Stability of OLE2 vs. CORBA*** End-user system life cycles involving custom software development typically range from 10 to 15 years. This range defines the practical life span of a technology. Commercial software technology works on a more compressed timeline; new products and major upgrades are introduced to the market at six- to 12-month intervals. The commercial viability of a major technology can be as short as one to three years before a new wave of innovation changes the market. In order to remain competitive, commercial vendors must begin development well in advance of the current state of the art. In markets undergoing revolutionary changes (such as CORBA and OLE/COM), marketing staff must recruit an enthusiastic group of early adopters to ensure independent software vendor and end-user support. The marketing staff must also educate independent vendors and users in the mainstream market about the revolutionary benefits of the new technologies, lobbying mainstream organizations to align their strategic plans to adopt their technologies as they are delivered to market in productized form.

OLE/COM is an excellent example of this process in action. There is an increasing number of OLE2-compliant commercial software due to the 80%+ desktop market share controlled by Microsoft. Microsoft is preparing its market for a revolutionary shift to new operating system technology, through the transition platform of Windows 95 and then to Cairo. Microsoft is a major independent software vendor in its own right and has used its

internal products as the early adopter group to support OLE2 with the Microsoft Office products. Through conferences Microsoft has been working to educate external developers about the new technology. It also has issued press releases and worked with magazines and other publications to reach the mainstream market with its message.

Given that competing technologies, such as OpenDoc, Taligent, and Fresco, are being released to the market with technology advantages over OLE2, Microsoft must advance its strategy to a new generation of technology within the next two years. A key problem for developers is that rapid innovation causes API obsolescence. An end-user system that adopts OLE2 with a 10- to 15-year life cycle would be assuming an extreme risk. If as is likely, Microsoft upgrades or replaces this API in two years, there will be significant operating and management costs to pay in end-user systems in order to continue to keep pace with Microsoft's rate of innovations. Upgrading all independent vendors' software may be necessary as well. Microsoft is attempting to provide transition support with backward compatibility from Windows 3.1 to Windows 95.

An important factor that moderates the pace of commercial innovation and technology obsolescence is standardization. Standards are a widely used marketing strategy; the cost of most standards activities is an investment provided by for-profit companies. Standards reduce risk for both suppliers and consumers and bestow credibility and acceptance on new technologies. Standards allow common infrastructure to be established that vendors and end users can leverage to support more platforms and more functionality. Formal standards creation through national and international bodies takes at least four years. Voluntary industry groups (such as OMG) can establish major standards much faster.

Once standards are established, the standard extends the life cycle viability of a technology (Figure 6.6). In the case of layered technologies (such as networking), standards can extend the viability of a technology indefinitely. The OMG has established a hierarchy of technologies that are layered upwardly in its Object Management Architecture [OMG, 93]. In the OMG model, the Object Services are layered on CORBA, and the Common Facilities are built on the Object Services.

Due to its multivendor support, standards basis, and layered architecture, CORBA has all the indications of being a technology with a long life cycle. Its life cycle continues to extend as other standards bodies increasingly utilize CORBA as the basis for standards (such as the International Standards Organization's Open Distributed Processing, X/Consortium, X/Open, etc.).

### **THE OPENDOC FRAMEWORK**

OpenDoc is a framework supporting innovations for the end user (compound documents) and the developer (component software). OpenDoc is a multivendor technology destined for standards support. For the user and developer,

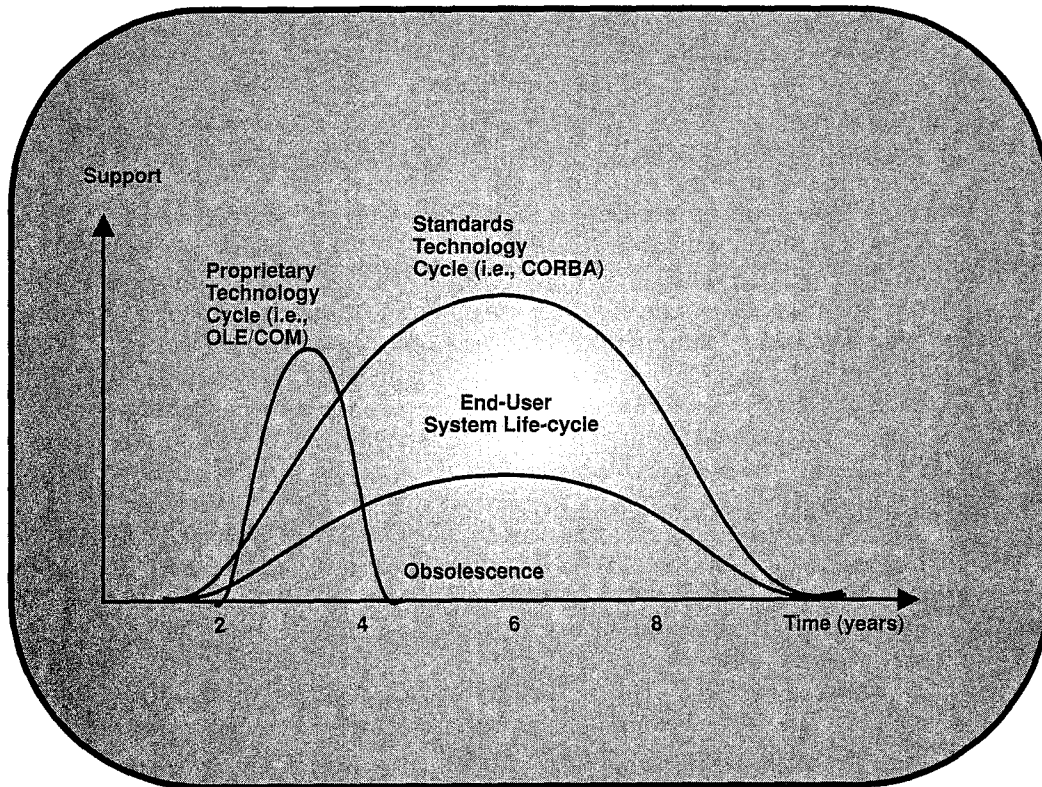


Figure 6.6. Standards and technology cycles vs. system life cycles.

OpenDoc offers some fundamental innovations in technology, representing the next generation of the end-user desktop and the application development environment. We begin our examination of OpenDoc by explaining the user interface model and how it differs from previous technologies.

### OpenDoc User Interface

Today's application-based desktops force end users to work in restricted modes. Each application has a unique data format and a unique set of commands for manipulating that format. The command set is restrictive in terms of manipulating external data formats. In order to manipulate information in another format, the end user must move to another application, with similar restrictions. Most end-user document products involve more than one form of data. If data is combined through a clipboard data exchange, then data fidelity or the ability to edit the data may be lost. If data is combined through linking, applications still have to be switched to manipulate linked data. Linking is very brittle in today's desktop technologies such as OLE2.



End users spend a great deal of time and energy coping with the constraints of the application-centered desktop model.

For application suppliers, the trend has been to create very large monolithic applications with increasingly complex sets of controls. Since each application must be fully independent and do as much as possible, suppliers are competing with each other to create the largest, most complex monolithic applications. Suppliers have the difficult problem of maintaining these complex applications on multiple platforms (which are also evolving). This increasing complexity is having an adverse impact on the end user, who typically uses less than 15 percent of an application's functionality and must keep upgrading training and hardware to accommodate these trends.

With the latest step in monolithic application integration, OLE2, end users become even more constrained because they lose their ability to transport documents between machines. OLE2 documents are heavily dependent on the local set of applications and the local pathnames of applications and data objects. In OLE2, large monolithic applications are present; they still require large memory resources and time delays for activation. The primary innovation in OLE2 is that the container application can present the user interface of another large monolithic application. OLE2 has not fundamentally changed the desktop paradigm; rather it presents it in a slightly different way.

OpenDoc changes the end-user interface fundamentally, from focusing on applications to focusing on the end-user's document products. OpenDoc refocuses the desktop on the end user's document products and completely eliminates the notion of monolithic applications. Documents are containers that can contain any types of content (Figure 6.7). The container itself is content neutral, and global functions, such as Undo, are transparently coordinated among the parts.

Each type of content can be manipulated by associated software called a part. Parts have sample documents called stationery. To insert a new type of content, the end user drags the stationery into a document. The user can drag and drop content between documents, creating a new copy or a linked copy under user control.

OpenDoc containers support a standard pair of menus: Document and Edit (Figure 6.8). The Document menu replaces the traditional File menu. Because OpenDoc eliminates the concept of separate applications, there is no quit command; documents are opened and closed instead of applications being launched and quitted. The Drafts command supports a version control capability built in to OpenDoc. Users can create multiple drafts of a document, which can be retrieved and edited later. The Mail command indicates that all OpenDoc documents will be mail-enabled. A parts bin is available, similar to the scrapbook, which contains reference objects.

The Edit menu is similar to conventional Edit menus (Figure 6.8). The Undo command applies globally across all parts within the document. This

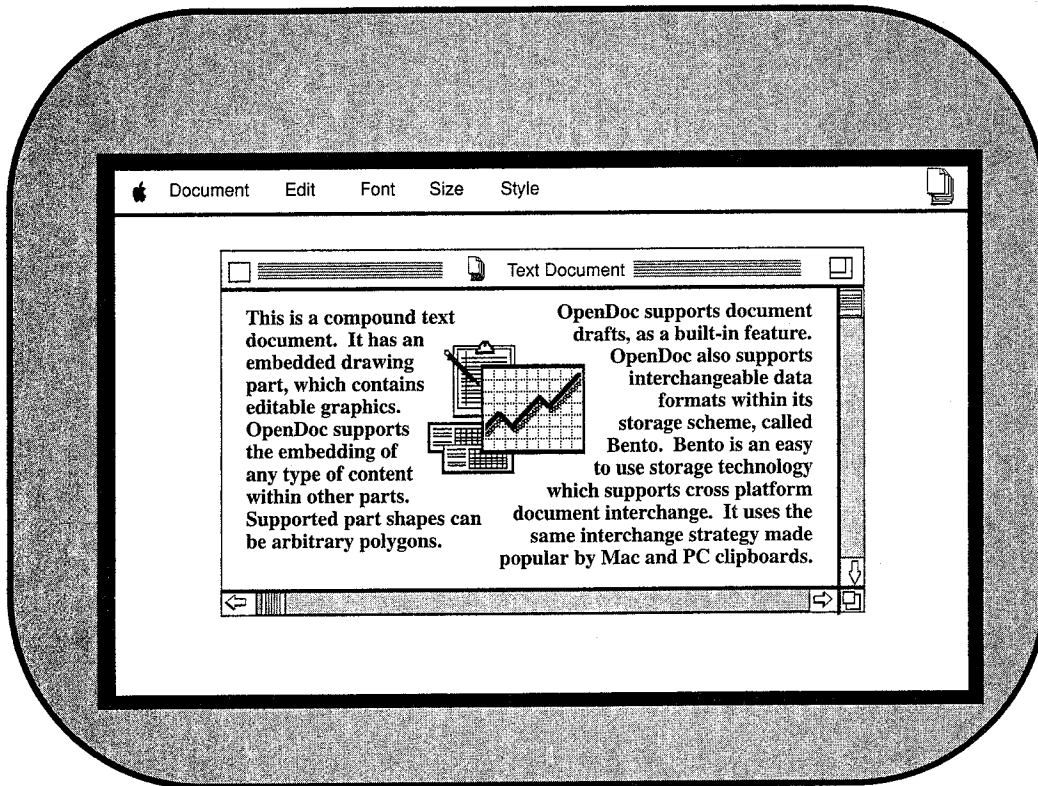


Figure 6.7. OpenDoc user interface.

is a powerful capability for undoing actions, regardless of the responsible part editor. The Cut, Copy, and Paste commands manipulate the clipboard. The end user may utilize drag-and-drop operations to perform these functions. The Paste As command supports active linking. The View as Window command allows any part to be instantiated within a separate window. This is convenient for some end-user editing operations.

OpenDoc documents are stored in interchange formats, similar to today's PC clipboards. Stored formats are platform independent and identified with registered format descriptors. This allows documents to be easily relocated, viewed, and edited by alternative software, depending on the platform. OpenDoc links, implemented through CORBA objects, can be translated across machine boundaries in a distributed network. These features allow compound document relocation, an important capability not found in other technologies, such as OLE2.

OpenDoc will be fully interoperable with OLE2. OpenDoc parts can be embedded in OLE2 documents, and OLE2 components can be embedded

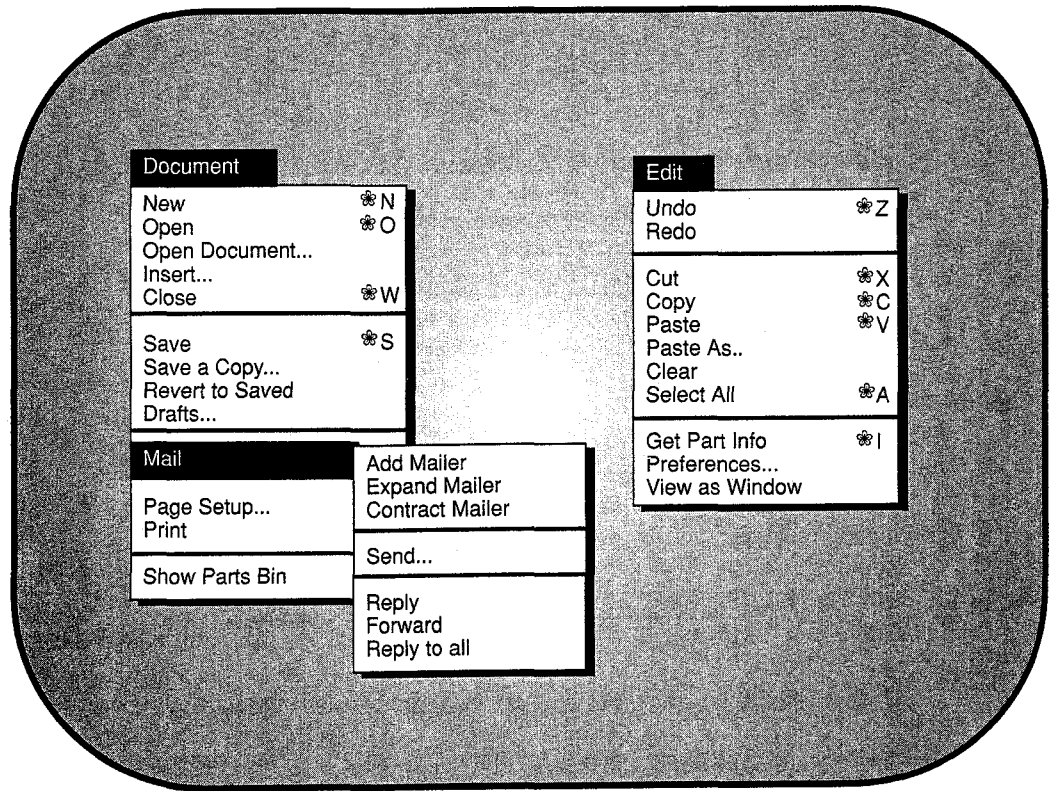


Figure 6.8. OpenDoc menus.

in OpenDoc containers. OpenDoc offers a simpler way to integrate OLE2 capabilities than direct OLE2 integration. It also offers other advantages, such as a more flexible embedding model and distributed support through CORBA.

Parts also have part viewers, which suppliers generally give away. Part editors and part viewers are much smaller and simpler than today's monolithic applications. Parts can provide specialized functionality without having to provide a complete stand-alone application around it. Suppliers can focus their resources on part editors that manipulate specialized content, instead of competing with all other monolithic applications that replicate a great deal of each other's functionality.

### OpenDoc Technology and CI Labs

OpenDoc is a technology administrated by a nonprofit consortium, the Component Integration Labs (CI Labs). Sponsor members of the CI Labs include Apple, Borland, IBM, Lotus, Novell, Oracle, Sun, Taligent, WordPerfect, and

Xerox. Several sponsors have made substantial corporate commitments: Apple contributed foundation technologies; IBM contributed CORBA implementation; and Novell is developing the Microsoft Windows implementation and the OLE2 interoperability solution to OpenDoc. Other sponsors are committed to providing OpenDoc integrated with their products and available for development on their platforms.

CI Labs ensures that the OpenDoc technology is vendor independent. It maintains the OpenDoc source code, disseminates development kits, and documentation. It also pursues testing, registry, and standards activities.

One of CI Labs' key roles is OpenDoc parts testing, called the validation service. For a fee, CI Labs will validate that a supplier or end user's OpenDoc part is compliant with the OpenDoc specifications. Compliance will assure cross-platform portability. CI Labs will maintain a hardware suite of all the available OpenDoc platforms and special versions of the OpenDoc implementation to assure independence from underlying platform implementations.

CI Labs maintains a registry of OpenDoc usage conventions that will assure interoperability between parts implementations. One of the registries is a list of categories of OpenDoc content types. Example categories may include text, tables, and graphics. These categories organize the registry for other registered items, such as storage data formats and predefined scripting events. System software can use content types to associate locally available parts editors with parts data. Registered data formats will be transportable across platforms and viewable/editable by a wide range of software.

CI Labs coordinates OpenDoc standards activities. CI Labs and OMG are developing a standard compound document framework based on OpenDoc. The adopted specification will be a language- and platform-independent compound document standard that enjoys strong international support.

Technologies underlying OpenDoc include the System Object Model, Bento, and the Open Scripting Architecture (OSA). These technologies are licensed to CI Labs for redistribution.

SOM is a CORBA-compliant object request broker. It is available across a widening range of PC, UNIX, and mainframe platforms. Originally developed for the PC-class OS/2 operating system, it supports the low-overhead efficiency needed for PC desktops. SOM enables transparent use of multiple programming languages with OpenDoc (initially C, C++, and SmallTalk). With additional CORBA language mappings, virtually any language can be integrated to OpenDoc transparently. SOM interoperability with other request brokers will allow OpenDoc to operate transparently across a network, supporting functions such as distributed linking (Mosaic-like hypermedia), remote part editors, and multiuser collaboration. SOM and OpenDoc will be integral elements of Copeland, Apple's next-generation operating system.

Bento is a compound document storage specification. It is a very flexible and simple technology that supports storage of any content type and is platform and media independent. Bento handlers, which perform the low-

level storage manipulation, can implement device specific buffering, packing, and encryption. The device handlers are transparent to the part editor code.

The OSA supports interpart communication in OpenDoc. OSA will enable groups of OpenDoc parts to work together in documents. For example, a group of parts may want to choreograph an animated display of information in a document. OSA combines a high-level messaging facility with a common scripting language and a common event format. OSA supports the consolidation and abstraction of events by the operating system, called semantic events. For example, instead of reporting a low-level mouse action (MouseUp, pixelX, pixelY), OpenDoc's semantic events might report: ("Copy" Command, Cell D10, "July 94 Report" Spreadsheet).

OSA supports three forms of interoperability with scripts. *Scriptable* parts are those that can receive commands via OSA scripts and events. *Recordable* parts are ones that can create reusable scripts by recording end-user actions. *Tinkerable* parts are parts whose behavior can be modified through the attachment of scripts by external applications. CI Labs envisions a large third-party market for script-authoring packages and intelligent agents that can create and execute multiapplication scripts.

### The OpenDoc Architecture

Using the OpenDoc architecture, the developer is involved in creating the *grand illusion* of direct manipulation. To the end user, it appears as if the mouse is hard-wired to the on-screen sprite, and the user can drag and drop desktop objects as if manipulating real-world objects. Behind the scenes, direct manipulation is implemented entirely using software. OpenDoc and the operating system implement the lower-level details of the grand illusion. OpenDoc presents a much-simplified abstract programming interface for the application-level software developer. Its developers can create new direct manipulation applications with less software than previous technologies required. OpenDoc has made it easier for developers to implement the compound document model, where the granularity of direct manipulation is at the embedded part level instead of the file level. In addition, OpenDoc software is a portable cross platform, is interoperable with multiple languages, will automatically support OLE2, and supports distributed processing. The potential advantages for the developer and end user are substantial.

OpenDoc is a sophisticated compound document architecture, comprising several dozen object types. Most of this complexity is transparent to the developer. Almost all of these object types are already implemented by OpenDoc. Fortunately, OpenDoc part developers are concerned only with the implementation of one object: their own part. Simple parts might interact with a handful of OpenDoc objects. Complex parts might interact with a dozen or so other OpenDoc objects. It's all a matter of how much of OpenDoc functionality a part supports. Functionality can start out simple

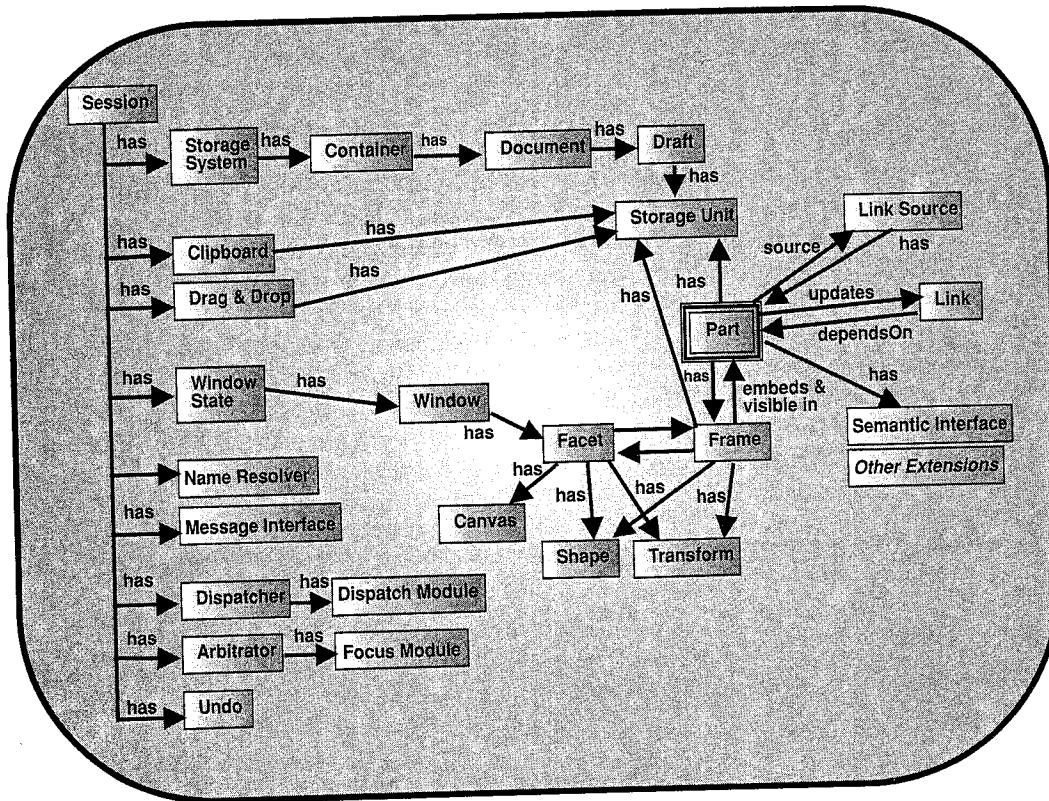


Figure 6.9. OpenDoc object relationships.

and grow incrementally with the part. Initially, storage, controls, and display are the most basic functions. More advanced capabilities include the clipboard, linking, and drag and drop. Advanced capabilities include embedding and scripting. Implementation of object embedding within a part is an advanced function that not every part will support.

OpenDoc provides support for creating a comprehensive template for the part software, through a point-and-click application called PartMaker. PartMaker creates a complete set of source files, header files, and makefiles. These files support default OpenDoc functionality, and the templates contain the entry points for every potential capability of an OpenDoc part.

One of the key objects in the OpenDoc architecture is the Session object. There is one session object for each OpenDoc document. Parts use the session object to locate other OpenDoc objects. For example, a part needs to interact with other OpenDoc objects that manage desktop resources and provide other OpenDoc functionality. The Session object implements messages such

as `GetClipboard` and `GetLinkManager` to retrieve the object reference of the requested object.

The primary presentation objects in OpenDoc are `Frames` and `Facets`, which support printing and display. A `Frame` is an area of a document allocated to a part. A `Facet` corresponds to any displayable area in the frame. Conceptually, we can think of a `Facet` as a lens through which the user can view a `Frame`. `Facets` are created by OpenDoc automatically when a frame becomes viewable. The part is notified of these events through various callback entry points. Parts may have more than one frame; there also can be more than one facet per frame.

The presentation objects work together in the OpenDoc framework to simplify the drawing model of the part software. `Frames` and `Facets` have associated shape objects that define their geometric extents. Shape objects are used to constrain the frame shape of the document, the clipped shape of the displayable facet, and the shapes of objects embedded in the part. When the part's `draw` method is called, the part can draw to the allocated frame

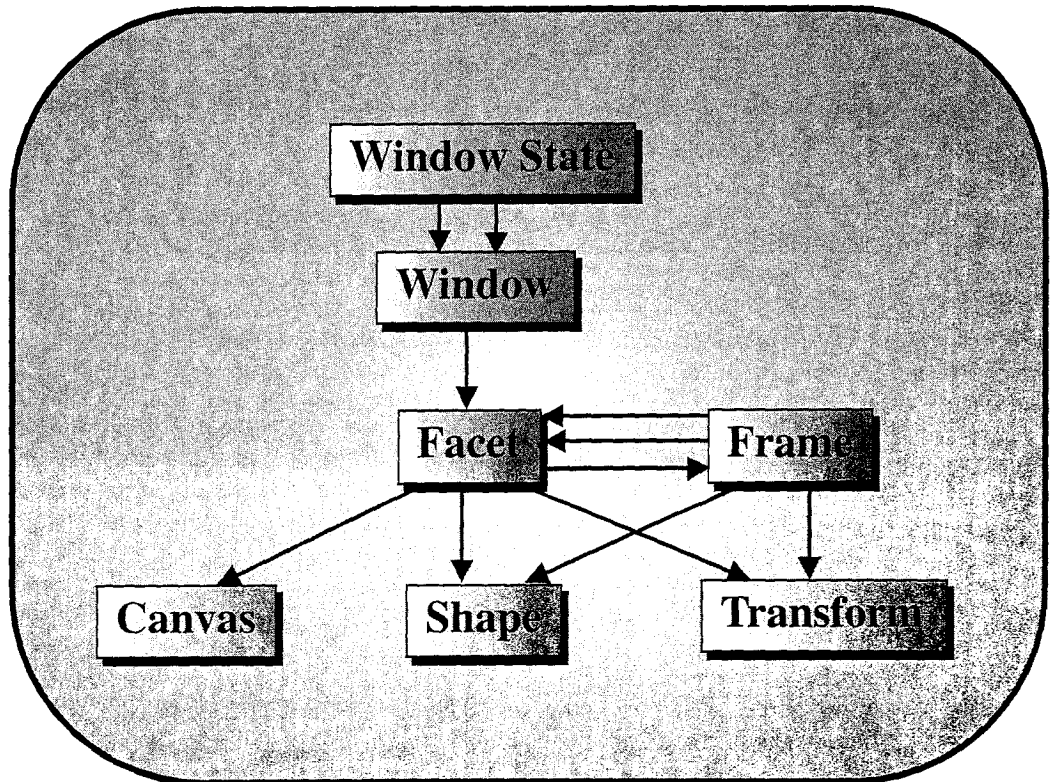


Figure 6.10. OpenDoc windowing objects.



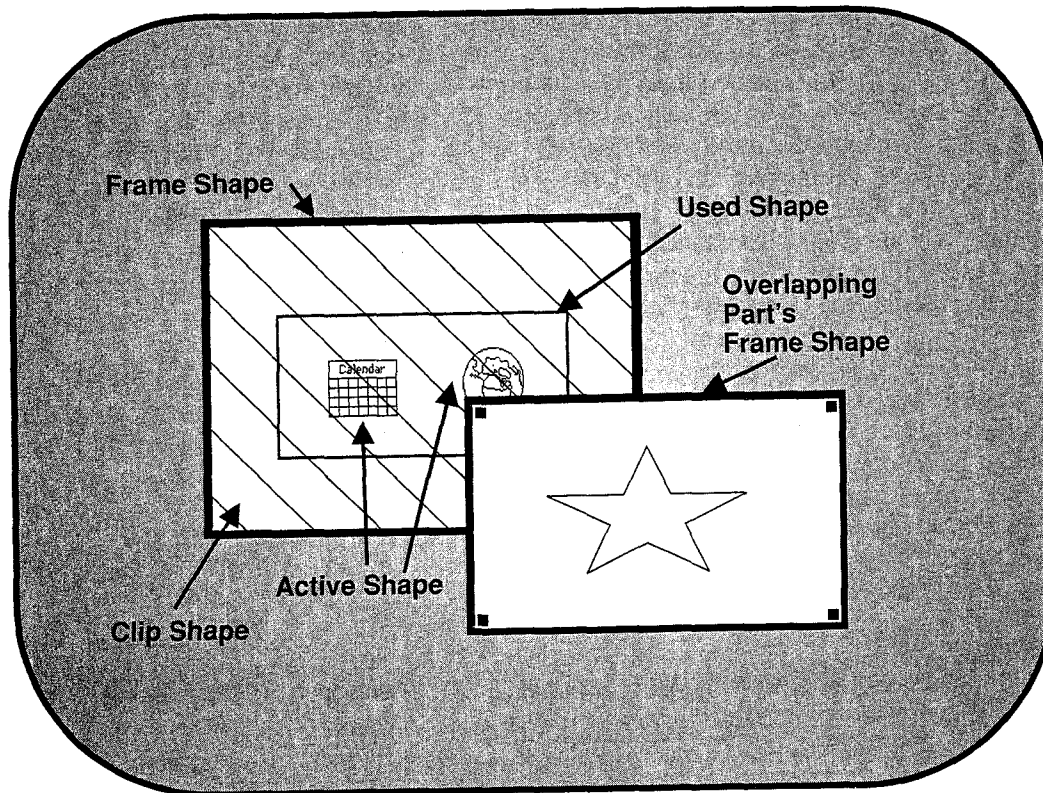


Figure 6.11. OpenDoc clipping.

shape, and the displayable clip shape can be applied transparently. The part can then invoke the draw method on embedded parts to redraw contained areas. Transform objects are used to indicate the location of a frame or facet within a containing part and the amount scrolled for scrollable parts.

Events are a key feature of OpenDoc available for use by developers. An OpenDoc part receives events through a `HandleEvent` method invoked by the Dispatcher object. Events identify the Frame, Facet, and self-descriptive event information, such as the event type, time, location, and option keys. The Dispatcher object filters platform-dependent events, so that parts perceive that events are handled in a platform-independent manner.

OpenDoc has a set of event focuses (i.e., keyboard, mouse) that must be allocated by parts in order to receive events. Ownership of event foci determines the distribution of an event. The Arbitrator object manages the allocation of foci. Some examples of event foci include the keyboard, the mouse, and the menus. For example, a part may obtain the keyboard focus from the Arbitrator and then receive subsequent keyboard events. Parts generally



request a complete set of foci when they are activated. OpenDoc provides a complete protocol for focus acquisition, preemption, and relinquishment.

Parts that obtain menu focus may modify the menubar to reflect part-specific commands. This straightforward procedure includes adding new menus and commands to the base menubar. The new menu structure can be assembled offscreen and displayed immediately after obtaining menu focus. To make the menu commands localized (for internationalization), the menu command titles should be retrieved from stored resource files. The part developer also is responsible for enabling and disabling menu items. A part is notified to adjust the menus just prior to menu selection by the OpenDoc WindowState object.

Storage units are the basis for persistence and data interchange in OpenDoc. Every part uses a storage unit to save and retrieve its persistent state. The same storage APIs are the basis for all forms of OpenDoc data interchange, including the clipboard, drag and drop, and linking.

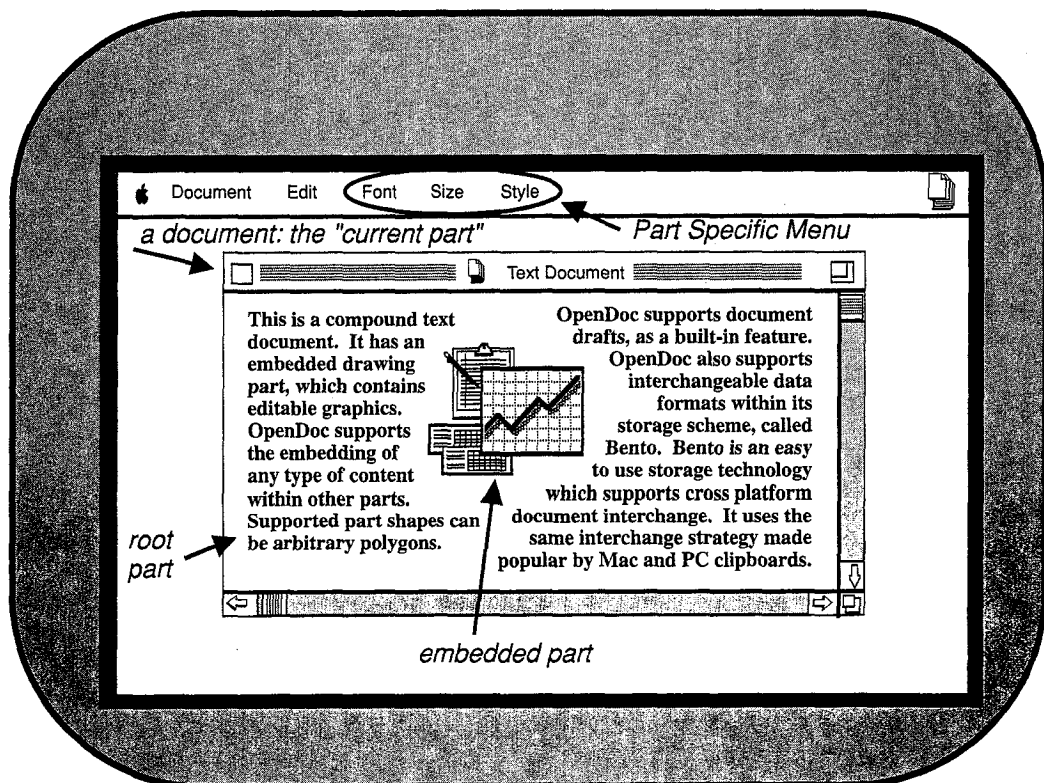


Figure 6.12. OpenDoc user interface parts, containers, and menus.

OpenDoc storage units are based on the Bento storage specification. Technically, Bento provides a simple API and storage structure that replaces conventional file input/output. Bento containers are a sequence of named properties. Each property can have multiple named representations. The data representations are byte sequences. In order to store structured data or pointers, the part must convert the data to a byte sequence before writing to the storage unit. This data structure flattening operation is called externalization. Conversely, the part must internalize the data to reconstitute the information when initialized from the stored state. OpenDoc calls the part's `Externalize` and `InitPartFromStorage` methods to request these actions.

Bento containers typically are implemented as files. Each container can store a complex compound document, comprising data from multiple part editors. This replaces the need for multiple files, created by separate applications.

OpenDoc parts typically store information in multiple exchange formats, similar to a clipboard. The part can store its highest-fidelity data format, followed by one or more common exchange formats. This feature of OpenDoc greatly increases portability of documents; other compound document technologies (i.e., OLE2) have limited ability to move documents between machines.

OpenDoc parts must keep track of lists of several important items, including the frames it supports, the objects it embeds, and active links with other parts. These lists should be stored in the object's persistent state so that they can be retrieved upon activation. A reusable collection class for iterating through these lists is useful for managing this process.

The OpenDoc Clipboard is an object created by the Session object. The Clipboard is used to interchange data, as on PC desktops. Before the Clipboard is accessed, it must be locked for exclusive access. The data on the Clipboard is in an OpenDoc Storage Unit. Parts can monitor changes to the Clipboard through the retrieval of a Clipboard version key.

The Clipboard supports data interchange in multiple fidelities. This is the same strategy used by current PC and Macintosh clipboards. The parts write their highest-fidelity native format, followed by other interchange formats. For example, a word processor part might write native format, Apple-styled text (a popular interchange format), plain text, and a link specification. This last piece of information would support active linking if the end user creates a link while pasting the information. Link objects also use Storage Units to transfer information.

The final use of Storage Units is in support of Drag and Drop. Drag and Drop is one of the most complex protocols for desktop objects. OpenDoc has abstracted the process to its essentials. The following is a brief synopsis of a typical Drag and Drop operation.

To initiate a drag within a part, the part receives a mouse event and must determine if it refers to a specific selected object (i.e., a drag is initiating); otherwise the mouse event is indicating some other operation. The part can then write to data to the DragAndDrop object's Storage Unit. The part then calls the StartDrag method of the DragAndDrop object. The balance of the Drag and Drop protocol involves the recipient of the Drag and Drop.

If a part is the recipient of a Drag & Drop (indicated by an OpenDoc call to the part's DragEnter method), then the part can determine if the appropriate format resides in the DragAndDrop object's Storage Unit. If so, the part can highlight itself (indicating potential acceptance); otherwise it can ignore the drag operation. The part can receive one or more DragWithin method calls during this process. The part receives a call to its Drop method if the user releases the mouse button; then the part reads the data, as if reading the Clipboard's Storage Unit.

Embedding is the most complex aspect of OpenDoc programming. Embedding is the ability of a part to contain other parts. OLE2 uses a single-layer embedding model with nonoverlapping rectangular frames. An OLE2 application implements either a container or an embedded component, but not both. OpenDoc embedding capabilities are much more flexible. Embedded shapes can be arbitrary polygons; embedded objects can overlap. Objects can embed within other objects to any level of nesting.

This complexity is hidden from the part developer who does not support embedding of other parts. The container object must deal with the unusual shapes of embedded objects and these other issues. When an individual part supports embedding, it must also contend with these issues.

## **AUTONOMOUS LAND VEHICLE**

An interesting example of a framework-based development occurred on an Advanced Research Projects Agency (ARPA) project called the Autonomous Land Vehicle (ALV). The research was conducted at the Martin Marietta Corporation (MMC) from 1985 through 1989. The design of the architecture predated CORBA but utilized many of the architectural principles presented in this book. Follow-on projects to ALV are being actively pursued at MMC and at many universities and contractors.

The ALV technical challenge was to develop a rapid prototyping testbed for intelligent mobile robotics research. The testbed supported the rapid integration of technologies from four research fields: image understanding, artificial intelligence route planning, sensors, and experimental parallel processors. These technologies were supplied by external research groups at universities and contractor organizations, evolving their research prototypes in parallel.

To prove the testbed capabilities and provide a general integration framework, MMC first developed and demonstrated its own set of algorithms. The

first experiment included continuous-motion road following. This demonstration was beyond the state of the art of what had ever been done in the research community; each run of the vehicle over its track included processing of over 700 video images. Prior to this experiment, the image understanding research community had focused on a small number of test images and had not demonstrated continuous-motion robotics. By processing so large an image set, ALV fundamentally changed the nature of the research problems.

Early ALV experiments focused on this continuous-motion road following. Later experiments demonstrated increases in road following speed using ARPA's advanced parallel processors. Other advanced experiments shows obstacle avoidance with continuous motion and off-road cross-country navigation.

A key function of the testbed was the hosting of external research software. Each research group had a particular focus, such as the vision or route-planning aspects of the mobile robotics problem. It also had particular hardware and operating system dependencies; some ran on VAX/VMS, on VICOM VDP, on Sun workstations, on Symbolics, on ARPA/CMU/GE Warp Machines, and so on. The ALV testbed supports a highly heterogeneous environment, including, in addition to the preceding machines, Multibus I chassis, VME chassis, the ARPA/BBN Butterfly Machine, and potentially any other commercial or research platform. In addition, programming language support requirements ranged from assembly languages, to Pascal, FORTRAN, Lisp, and C. ALV needed to support all these heterogeneous requirements in a distributed laboratory, where different parts of the processing could occur potentially on any of the platforms. The ALV laboratory was distributed between the onboard processors and the laboratory facility, a conventional computer room environment. The vehicle and computer room were interconnected through a private FCC-licensed radio and TV station, which provided two-way digital data transfer and video transmission back to the laboratory.

ALV would be a challenging system to implement today; in the mid-1980s it required an extraordinary architectural vision and discipline to create a system flexible enough to cover all of these platforms, languages, and distribution options.

The ALV solution was to create a flexible integration framework that provided a consistent access mechanism for communication between distributed processes, called Real-Net. For the application programmers, Real-Net included a simple messaging API that provided transparency of location, platform, and language. Individual components of the ALV software could be replaced and/or relocated without impacting other elements of the system.

Real-Net was implemented primarily over an Ethernet LAN connection, with Real-Net software communicating at the UDP level. Because of the mobile distributed environment, the Ethernet protocols also ran over the

digital radio link. Real-Net supported other hardware connections, such as real-time direct memory access (DMA) transfers between VME chassis. The choice of networking hardware and protocols was made transparent to the application software.

Application software communicated through a simple set of send-and-receive APIs. It provided the application-level parameters and abstract destinations. The parameter formats and network addresses of the messages were stored in tables that were initialized at system boot time. The details of the network addresses were hidden from the application programmers. This allowed the processes to be replaced and migrated without changing application software. The Real-Net architecture provided native language and operating system APIs on each of the heterogeneous platforms. Part of the system was implemented in the native languages of the platforms involved. The commonality occurred at the network packet level, which could be marshalled and unmarshalled according to the platform and language-specific requirements.

With today's CORBA-based technology, ORB vendors provide a great deal of the Real-Net functionality as specified by OMG standards. CORBA provides transparent heterogeneous processing that is location, platform, and language independent. CORBA actually provides more flexibility, in that it binds processes at runtime and can establish new process allocations and relationships dynamically. Flexibility in the CORBA standard allows different kinds of underlying ORB implementations, including real-time and non-real-time products.

## COMMENTS

There is certainly no way to guarantee CORBA's continuing viability. Developers can approach CORBA in two alternative ways, and their choice will greatly influence the magnitude of the opportunity cost. The two approaches are: (1) use CORBA in a product-dependent manner for its ORB-dependent benefits, or (2) use CORBA in a technology-independent manner to define better software architectures.

If CORBA is used in a way that is heavily dependent on product-specific extensions to the standard, then CORBA failure will have a catastrophic impact. Some dramatic examples of this type of mistake became evident when HyperDesk withdrew its non-compliant ORB product from the market. A similar error might be made if an application system were developed to be highly dependent on OLE2, DCE RPC, or ToolTalk, since these technologies also might have short life spans.

Migration to CORBA should be approached with technology independence in mind. A primary goal of the migration should be the creation of an effective software architecture. The software architecture (specified in OMG IDL) should be designed to provide functionality in a cost-effective manner.

This can be achieved by selectively hiding, exposing, and abstracting the complexities of the subsystems. The software architecture should isolate the subsystems from each other and provide overall product independence so that subsystems can be replaced readily. It should support system extensibility by providing sufficient metadata and symmetry of representation to allow the addition of new subsystems without the need to modify existing subsystems. All of these features should be captured in the system's OMG IDL specifications. If one ORB product fails, the system can be ported to another ORB with minimum impact. If no ORBs are available, the OMG IDL APIs can be layered on top of an alternative mechanism, such as Remote Procedure Call (RPC). If this technology-independent approach is taken, the opportunity cost of using CORBA will be a net benefit, regardless of the success or failure of the CORBA standard.



## In-Depth Example: The DISCUS Framework

The Data Interchange and Synergistic Collateral Usage Study (DISCUS) developed the U.S. government's first CORBA-based application system. DISCUS has been recognized by technologists and the media as one of the best examples of the benefits of distributed object technology. For example, it was a finalist in the Computerworld Object Application Awards. DISCUS has been presented and demonstrated at more than a dozen national conferences. For software architects and developers, DISCUS is an important case study, demonstrating many successful strategies for systems integration using CORBA.

The DISCUS framework (without the demonstration system) was the primary technology product of the project. The framework comprises less than 150 lines of Object Management Group Interface Definition Language specifications (OMG IDL). For such a small amount of specification, it has been shown to yield some interesting and important results. The two key results include substantial interoperability benefits and low integration cost. The interoperability provided by DISCUS includes universal data interchange, data format conversions, universal data source access, and scriptable application control. This is achievable with a very minimal amount of integration code, averaging about 500 lines for each application. Compared to traditional integration projects, this represents more than an order of magnitude reduction in integration cost and complexity. Some of this gain was due to CORBA, but most of the benefits are due to good software architecture design coupled with a strong architectural vision shared by developers and program managers. CORBA and the OMG played a key role in DISCUS by clarifying the importance of good software architecture and the need for interface standards.



The DISCUS framework is unique in its simplicity. Only four operations form the core set for understanding the framework. DISCUS can be considered as the software analogy of Reduced Instruction Set Computer (RISC) microprocessor architectures. The DISCUS framework is simple and therefore easy for developers and organizations to learn and to use. Simplicity also reduces the integration complexity; less specification results in fewer application program interfaces (APIs) and less integration code.

DISCUS represents an architecture design point that balances integration cost and provides modest levels of generality and functionality. This design point might not be appropriate for all applications, but it seems to be a very good one for government software integration; we believe that it is applicable to the needs of many end-user organizations. The DISCUS framework presented here can be used readily by commercial and end-user organizations for integration projects.

In the following sections, we define the concepts, operations, and key services comprising DISCUS. The section on Framework Concepts presents the basic concepts of the architectural vision. Sections titled Application Objects, Data and Table Objects, and Factory Objects define the basic classes of the framework, their operations, and key design rationale. Framework Services describes some of the fundamental services supporting conversions, interchange, and metadata. Section, DISCUS Implementation covers important implementation approaches. The last section, DISCUS Issues and Futures, describes the key issues and futures for the DISCUS framework.

## FRAMEWORK CONCEPTS

DISCUS's architectural vision is to provide interoperability using a small set of common interfaces defined in OMG IDL. In general, interoperability is guaranteed if all subsystems in an integrated system share common interface and operations, as in a hybrid architecture. (See section titled "Software Architecture".) DISCUS is the minimal set of such operations that can be supported by different types of applications. The system is then extensible because new applications can be added (plugged in) without requiring changes to existing software.

Today custom interfaces are required to integrate most subsystems. Each subsystem usually requires a separate interface to each other subsystem. This is called the  $n \times n$  order interface solution (Figure 7.1).

If a single IDL interface is defined across the applications, the client can communicate with both implementations using a single set of operations (Figure 7.2).

The common interface solution allows more clients and object implementations to be added and to communicate with all existing and future applications (Figure 7.3). CORBA also supports the application-specific interfaces that may be published. Using OMG IDL subtyping, it is possible to define the application-specific interfaces as specializations of the common interfaces.

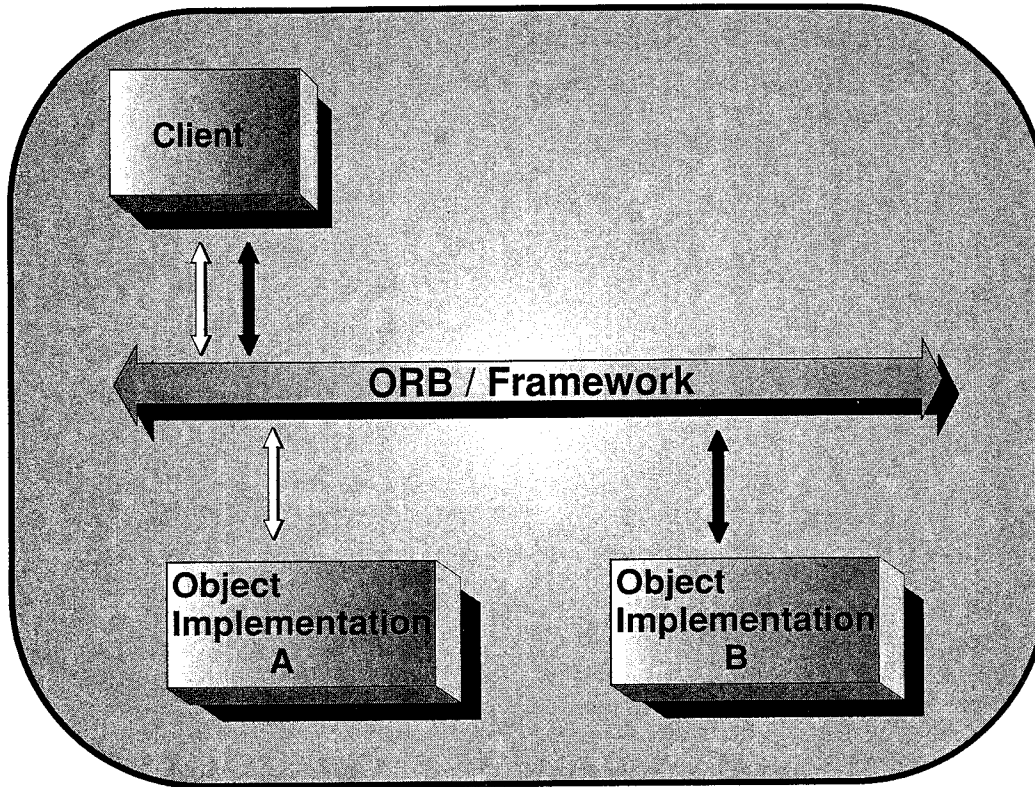


Figure 7.1. Client communicating with two object implementations with different IDL interfaces.

In order to facilitate interoperability between applications and between applications and data sources, the framework defines a set of operations for data interchange, query, data conversion, object wrapping, and encapsulation. The framework is made up of only the interfaces and a factory object; it does not define the implementation of the applications or the operations themselves. Different types of software applications can use these general-purpose operations. The operators can be defined as object-oriented specializations of a general application class. An application class can extend on the interfaces for communication with other members of the same class (e.g., Geographic Information System [GIS] mapping applications and data sources). However, the framework provides the minimal set of common operations that allow for interoperation with other types of applications without custom interface programming. In addition to the OMG IDL and factory, we also have implemented a set of framework services, such as the trader service and a conversion service. The infrastructure requires an underlying communication facility, such as a CORBA-compliant commercial object re-

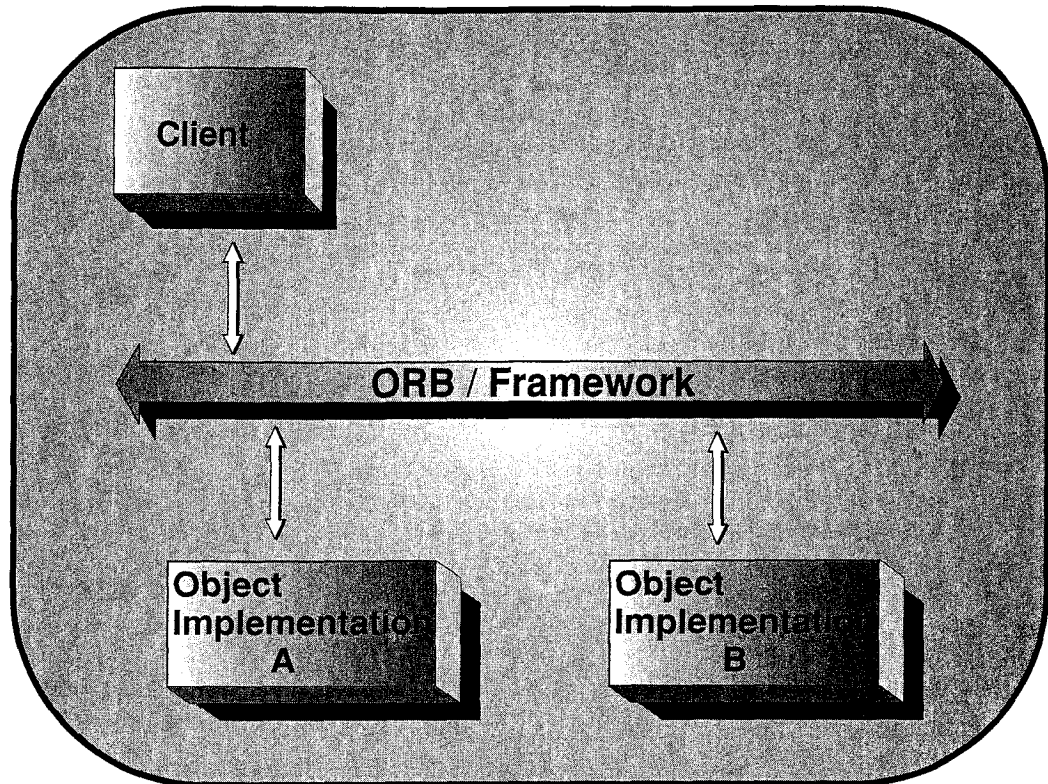


Figure 7.2. Client communicating with two object implementations with same framework IDL interface.

quest broker. Alternatively, it is possible to utilize an RPC mechanism as the underlying facility. However, if an RPC or other mechanism is used, there is a corresponding sacrifice in implementation flexibility and added coding.

At the highest level, the DISCUS framework comprises both application and data objects. The application objects are encapsulations of legacy applications, commercial applications, or new software. The data objects are generic containers for transfer of information. All of the applications, regardless of type, are encapsulated consistently with a common set of operations and consistent conventions for metadata. Communication between applications is in terms of the core framework operations that are used to transfer information encapsulated in data objects.

Four core framework operations provide the key benefits: interoperability, simplicity, ease of use, flexibility, and extensibility. These operations are:

- Convert
- Exchange
- Query
- Execute

These four operations provide the functionality of a whole host of complex specifications, from standards and proprietary sources. DISCUS is an application architecture that abstracts the interoperability solution to a very simple form. DISCUS is easy to learn and inexpensive to utilize due to its simplicity.

Extending from this simple core are a variety of design and implementation elements that complete the framework. Some of the key elements include table objects, factory objects, trader service, and object implementation metadata (described in the subsequent sections). The table objects are

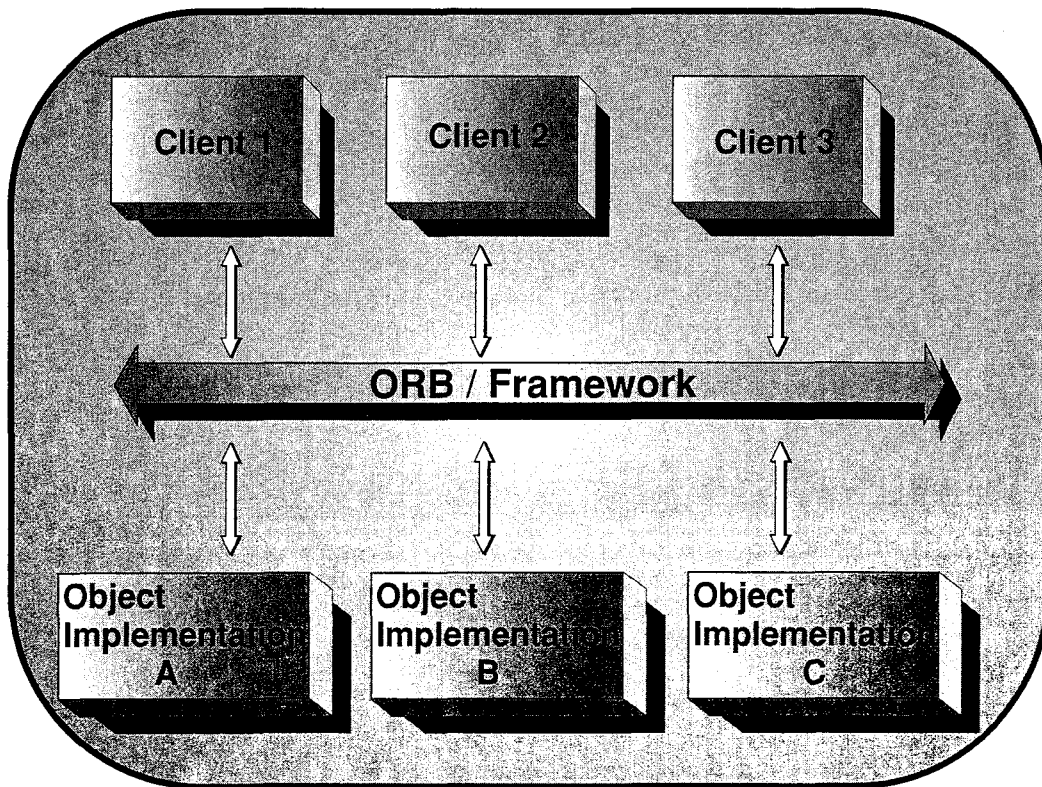


Figure 7.3. More clients and object implementations can be added without modification to existing software.

generic containers for tabular data. A tabular representation is very useful for interchanging metadata, query results, and other information. The factory objects manage the creation, copying, and deletion of data objects and table objects. The trader service manages systemwide metadata, including the location and basic capabilities of all services. The object implementation metadata comprises a more detailed form of object implementation-specific metadata, such as the object implementation's information schema.

## APPLICATION OBJECTS

### The Core Framework Operations

*DISCUS supports interfaces for basic interoperability through the use of the four core operations. The forms of interoperability supported include data interchange, access to data sources, and access to application functions.*

*Data Interchange* DISCUS supports interchange of most types of data used in a wide range of application domains. For example, the data can be images, maps, bitmaps, graphics, text, binary, and so forth. DISCUS provides a general-purpose data interchange facility, as opposed to a specialized facility. A specialized facility might optimize the interchange of a specific data type, such as a video stream, and could become a compatible extension.

*Access to Data Sources* DISCUS supports the retrieval of data from most kinds of information sources. The sources can reside on virtually any platform or be in the form of any database type, such as object oriented, relational, and flat files. DISCUS supports a general-purpose access to data sources, as opposed to a specialized or optimized access. By making data source access consistent, DISCUS simplifies the client programming for access to a wide range of data sources.

*Access to Applications Functions* DISCUS provides the ability to reuse the functionality of existing applications. Sequences of operations may be automated via scripts and may involve multiple independent application objects. DISCUS applies this form of application automation consistently across all kinds of applications using data objects to transfer arguments and results.

DISCUS is a working example of an application architecture constructed using CORBA. CORBA's flexibility can be used effectively with DISCUS to build highly interoperable systems. The DISCUS framework provides necessary structure to assure interoperability while providing sufficient flexibility to support the functionality needs for a wide range of application domains.

DISCUS purposely avoids addressing issues that are the subject of near-term standards from the OMG. Issues such as linking and embedding were delegated to the standards activities. DISCUS allows for the transitioning

to and leveraging of these technologies when they become available. The framework is flexible enough, though, to provide for similar capabilities now via other means.

DISCUS provides a fundamental level of guaranteed software interoperability. It is designed to be a reusable open architecture that does not define new standards but instead allows for use and leveraging of existing and future information systems standards. The framework defines two generic classes of objects: application objects and data objects.

Application objects include any type of applications, such as desktop applications, back-end servers, legacy subsystems, commercial software packages, and the like. Interoperability is guaranteed when these application objects implement the four basic DISCUS operations.

Data objects are used as containers for data interchange and results passing. Data objects can contain virtually any type of information. As simplicity for developers was a key goal of DISCUS, values are stored and retrieved using string-valued names with the simple `set()` and `get()` operations. The DISCUS factory controls the life cycle of data objects, their creation and destruction. The factory design is implementation dependent. The data objects may be passed between application objects, or the object references only may be passed and used as pointers to a data object stored possibly within a factory application object using a database.

***Convert Operation*** The convert operation is a general-purpose facility to convert data between various formats. The data formats may be imagery formats, document formats, spreadsheet formats, and others. The client provides the data with the current format and specifies the desired format. The implementation returns the data in the desired format, or an exception if the conversion failed. The generality of the operation enables each application (or vendor) to provide its own conversion(s). The trader service's metadata table for the convert operation allows each client to search for the availability of certain conversions and conversions may be added dynamically at any time. The client, or a smart conversion broker, may search the table for a series of conversions from format A to B. If no direct conversion is available, the conversion may be performed in multiple steps, from A to C and from C to B.

***Exchange Operation*** The exchange operation provides a simple general-purpose data interchange capability. Basic interoperability can be achieved if each application, at a minimum, implements the exchange operation. Applications can use the exchange operation to interchange data objects of any type and specify the type of exchange using a small number of enumerated options. The exchange may be used to transmit or receive data or to signal a request to open a front-end application on a data object. Clients also may use the exchange to request object implementation metadata. (See the

section titled “The Trader Service and Metadata Objects” on page 206). The object implementation metadata is a DISCUS data object that contains information regarding the object implementation data models and schema as well as operations and sample scripts that it may perform. Using DISCUS convenience functions, the client is able to utilize scripts without depending directly on the scripting language used by the object implementation.

**Query Operation** The query operation should be used to retrieve data when the desired data is not known ahead of time. The query server creates a new data object as the result of a query-driven search. A client may access an arbitrary data source using the query operation. It passes a script for the query server to perform, and the server returns a new data object if the query is successful, or an exception if it is not successful. The script is self-describing because a tag is attached identifying the query language (i.e., SQL89 Level 2). The script may be dynamically created, or it can be a predefined script retrieved with the server metadata. The server metadata object allows dynamic description of data sources, data types, and schemas. These facilities in combination can support almost any type of query accessing almost any kind of data source. New data sources and their query languages can be registered with the trader service to announce their availability. Each data source should be able to return a server metadata object in response to an exchange operation.

**Execute Operation** The execute operation is an automation facility that combines the features of the exchange and query operations for encapsulation of applications, such as legacy systems. The operation allows the passing of a script and a sequence of data objects to be operated upon. The object implementation modifies the data objects or creates new ones and can return a sequence of output data objects.

The following example best describes the possible use of the execute operation. Let's consider a legacy application that takes only command-line arguments and options and that can read two image files, register them (perform algorithmic operations that overlay the images and rotates them if necessary), and produce the output as a single new image file. The execute operation can be used to pass the commands in the script and the images within two image objects. The object implementation then can create the two image files from the display representations found within the data objects and execute the registration program using the script and file pathnames. Upon successful completion of the execution, the object implementation can create a new image object and encapsulate the resulting file into its display representation. The execute operation then returns to the client a sequence containing the single newly created data object.

The execute operation provides a structured way to access applications and utilities through scripts. It should be contrasted with the CORBA-



defined Dynamic Invocation Interface (DII), which provides dynamic access to applications via their available interface in the interface repository.

**OMG IDL Specifications for Application Objects**

All of the DISCUS objects inherit from an abstract class CommonObject. Its role is a mix-in class—a class that promulgates common definitions. CommonObject’s operations include open, close, and destroy.

All objects in the DISCUS framework implement the operations open, close, and destroy. The open operation is used to establish a client-to-object implementation session. Open allows the object implementation to set up any necessary resources. The close operation terminates a client-to-object implementation session. The destroy operation indicates that the object implementation should remove all associated resources for the session with that client.

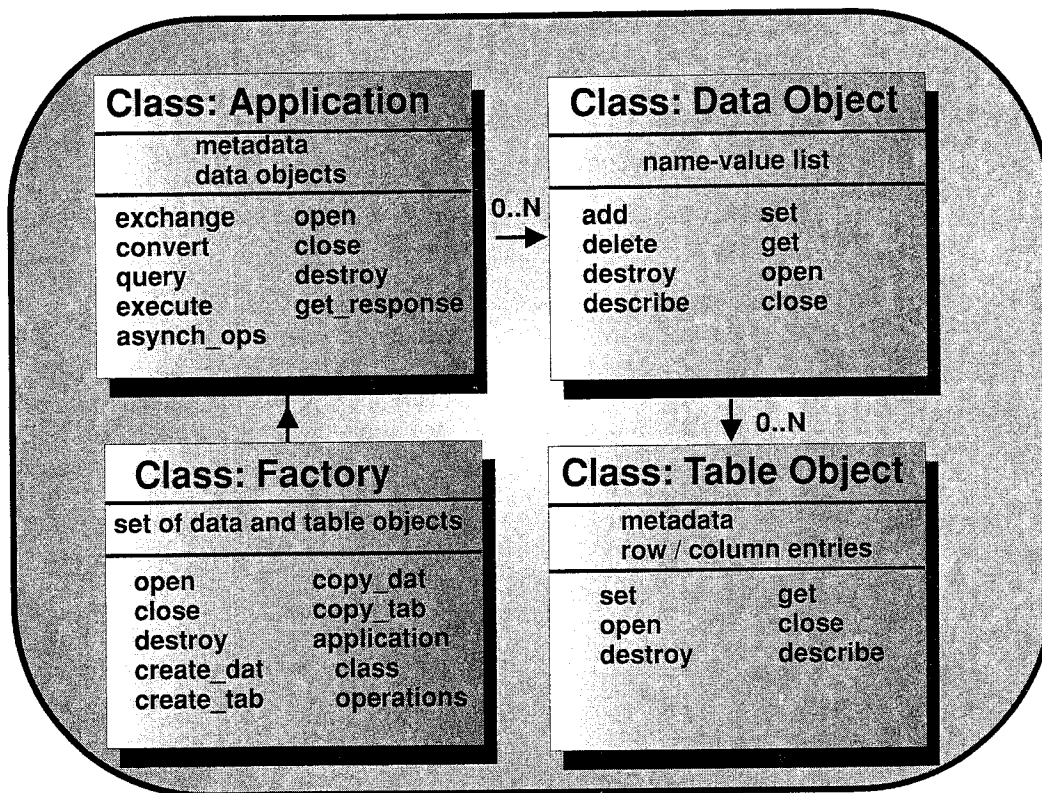


Figure 7.4. DISCUS class hierarchy.



```

// o-----o
// |           -- interface CommonObject --           |
// |           |                                     |
// | Abstract Class defining Common exceptions and    |
// | operations                                       |
// o-----o
interface CommonObject {

    //common exceptions
    exception ALREADY_OPEN DS_exception_body;
    exception NOT_OPEN DS_exception_body;

    // Delete all associated resources
    void destroy()
        context ( DS_context_attributes );

    // Establish Client to Object implementation Object Session
    void open()
        raises( ALREADY_OPEN )
        context ( DS_context_attributes );

    // Terminate Client to Object implementation Object Session
    void close()
        raises( NOT_OPEN )
        context ( DS_context_attributes );

}; /* end interface CommonObject */

```

The concrete interfaces of the DISCUS framework are contained in module DS, which comprises the Factory interface, the Data Object interface, the Table Object interface, and the Application Object interface. Each interface inherits from the common object. Polymorphism allows each open, close, and destroy to be implemented as appropriate to the particular interface.

```

module DS {
interface CommonObject {
.....
}; /* end interface CommonObject */

interface ap:DS::CommonObject {
.....
}; /* end interface ap */

interface dt:DS::CommonObject {
.....
}; /* end interface dt */

```

```

interface ft:DS::ap {
    .....
}; /* end interface ft */

interface tb:DS::CommonObject {
    .....
}; /* end interface tb */

}; /* end module DS */

```

The OMG IDL C mapping for the ft interface open is:

```
DS_ft_open();
```

Each interface represents a CLASS. Class application has a 0-to-*N* relationship with CLASS data object. Each application may reference zero or more data objects. Data objects, in turn, may reference zero or more table objects. The factory is a specialized application and inherits from the application class. It manages the life cycle of data and table objects.

The DISCUS operations are methods that should be implemented by each and all object implementations. Even if a specific operation is not supported, the method must return an `ex_DS_ap_UNSUPPORTED_EXTENSIONS` exception. The application object inherits the `DS_ap.open()`, `DS_ap.close()`, and `DS_ap.destroy()` from the Common object.

```

// 0-----0
// |                -- interface ap --                |
// |                                                    |
// | The DISCUS application Object. All DISCUS application |
// | interfaces inherit from this.                      |
// 0-----0
interface ap: DS::CommonObject {

    // DATA EXCHANGE OPERATION
    // This service should be supported by all applications.
    enum Operation { GETDATA,
                    PUTDATA,
                    OPENFRONTEND,
                    GETMETADATA };
    exception INVALID_EXCHANGE_TYPE DS_exception_body;

    void exchange (    in    Operation    exchangetype,
                    inout dt    dataobject )
                    raises ( NOT_OPEN, INVALID_EXCHANGE_TYPE )
                    context ( DS_context_attributes );
}

```

```

// FORMAT CONVERSION OPERATION
// Convert data from one format to another
exception INPUT_FORMAT_UNKNOWN DS_exception_body;
exception OUTPUT_FORMAT_UNKNOWN DS_exception_body;
void convert (
    in    string  format, // desired data format representation
    in    string  propertyname, // property containing input data
    inout dt      data object ) // object containing formatted data
    raises ( NOT_OPEN, INPUT_FORMAT_UNKNOWN,
            OUTPUT_FORMAT_UNKNOWN )
    context ( DS_context_attributes );

// QUERY/RETRIEVAL OPERATION
// Retrieve data from an application object based on some query
struct Script {
    string language; // query or script language
    string statements; }; // query or script statements
exception UNSUPPORTED_LANGUAGE
    DS_exception_body;
exception SCRIPT_SYNTAX DS_exception_body;
exception UNSUPPORTED_QUERY DS_exception_body;
exception UNKNOWN_OPERAND DS_exception_body;
exception INCOMPLETE_QUERY DS_exception_body;
void query (
    in Script query,
    out dt      responsedataobject )
    raises ( NOT_OPEN, UNKNOWN_OPERAND,
            UNSUPPORTED_QUERY,
            INCOMPLETE_QUERY,
            UNSUPPORTED_LANGUAGE,
            SCRIPT_SYNTAX )
    context ( DS_context_attributes );

// EXECUTE OPERATION
// This operation should be supported by all scriptable applications
// and processing services, i.e. any service not appropriate for
// Convert() and Query(). The processing is controlled with a
// script.
// Example "script"s:
// "sh", "csh", "perl", "AML", "CSL", "Tcl"
typedef sequence<dt> SeqObject;
void execute (
    in    Script      commandlist,
    in    SeqObject   inputdataobjects,
    out   SeqObject   outputdataobjects )

```

```

raises ( NOT_OPEN, UNSUPPORTED_LANGUAGE,
        SCRIPT_SYNTAX, UNKNOWN_OPERAND )
context ( DS_context_attributes );

}; /* end interface ap */

```

**Open** The application open operation should be called prior to the invocation of any other operation. The call establishes a connection with a particular object implementation and may be used by server implementations to set up any required resources.

The open is the first operation in which an object implementation receives the client's CORBA invocation context. The context may play a very important role in how communication and resources are set up for the rest of the session. In the context, the client may provide some information regarding the user's window system preference, communication-line capabilities, security information, and so on. Object implementations also may need to

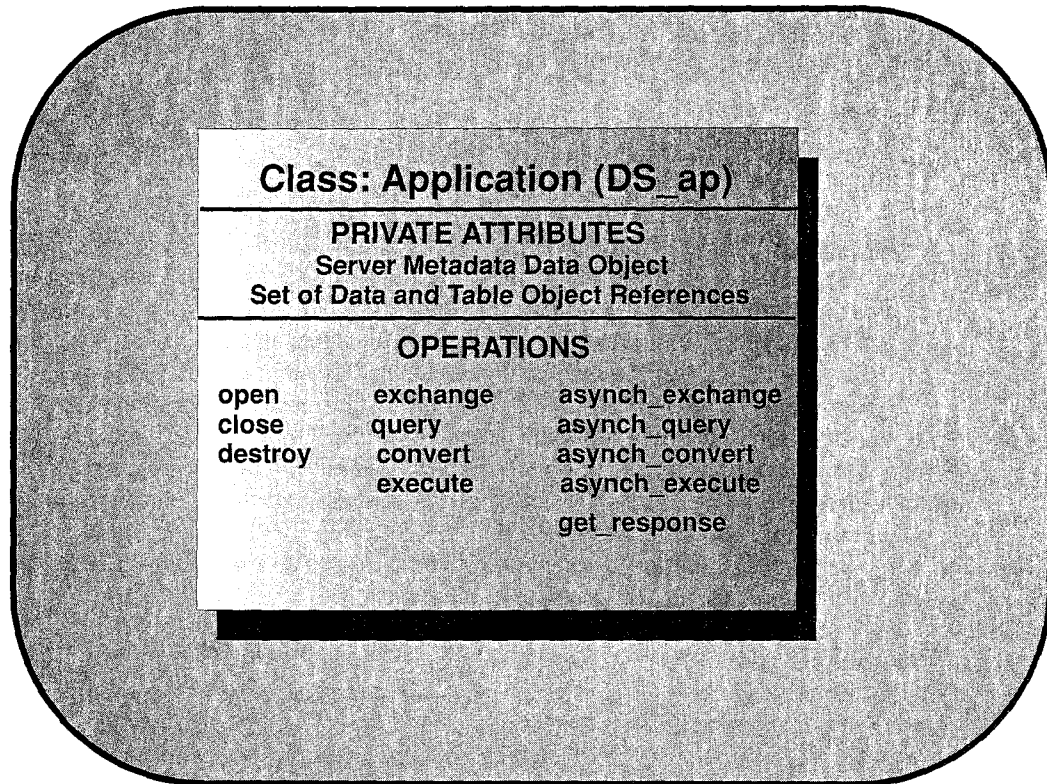


Figure 7.5. Application class definition.

retain certain client information if they are capable of supporting several clients at the same time.

**Close and Destroy** The application close operation is intended to signal to the object implementation that a session or logical set of operations have been completed. In some cases object implementations may choose to release certain resources at this point. The application destroy operation destroys all resources associated with a particular connection to an object implementation and signals the ORB that the connection with the particular object implementation should be terminated. The destroy should be called only after a **DS\_ap\_close** has been called. In other cases, where object implementations have some knowledge or understanding of the type of clients that they serve, resources may be released only using the destroy operation. The combination of close and destroy allows a client to signal the session status and give object implementations more flexibility to manage their resources. Otherwise, object implementations may have to wait until a time-out has occurred prior to releasing the resources. Use of time-outs to release resources can lead to problems, such as the unexpected invalidation of a bound object handle.

**Exchange** The exchange operation is provided to allow for the interchange of data objects when the data is known ahead of time, for example, as the result of a user selection.

In order to guarantee data interchange interoperability, the Exchange operation is the simplest and most restrictive of all the framework operations. By convention, all applications should provide client and object implementation support for the **exchange()** operation. An enumerated parameter defines the type of the interchange. The data object exchange can be from the client to object implementation only, from the object implementation to the client only, or from the client to object implementation and back. One of the enumerated types also supports the exchange of object implementation metadata information. This area should be supported to guarantee that clients can “discover” new object implementation capabilities dynamically and communicate with them.

The following example shows a typical invocation sequence using the **exchange()** operation.

- A client opens the framework factory.
- A client creates a data object of a particular type.
- A client opens the data object.
- A client sets the values of a data object as needed.
- A client closes the data object.
- A client opens a connection to an object implementation using open operation.

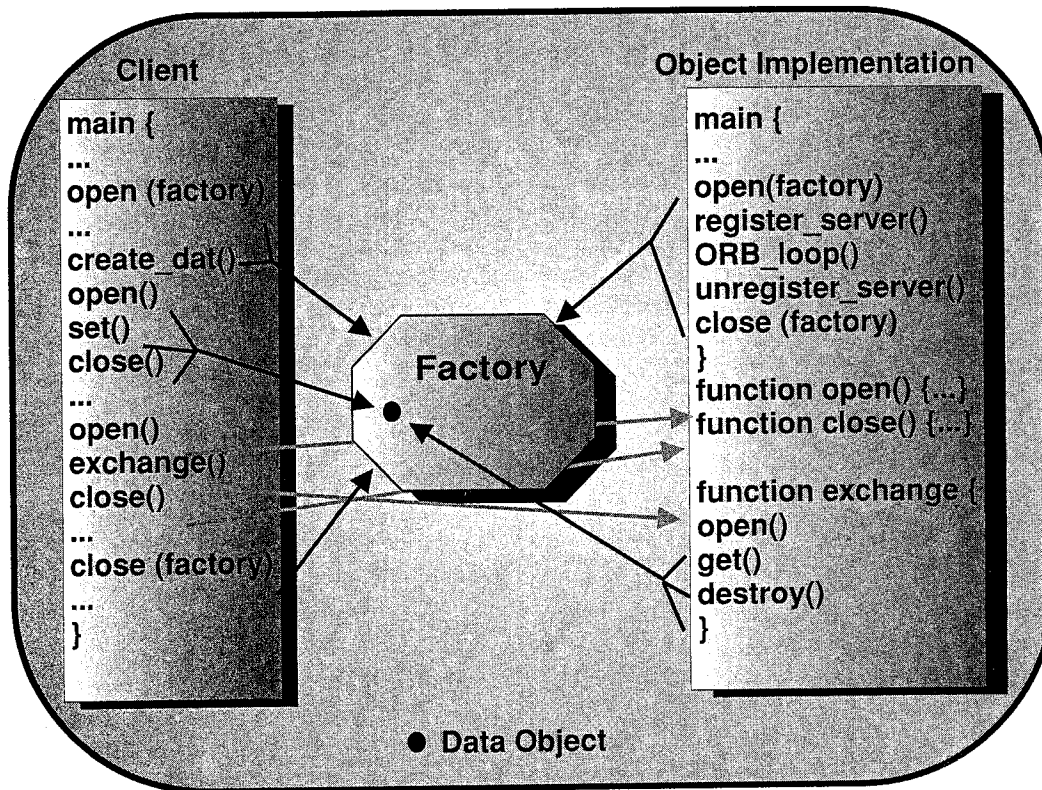


Figure 7.6. Exchange scenario.

- A client invokes an operation (exchange, convert, query, or execute).
- If the operation carries a data object reference, the object implementation may open the data object, get and set properties, and then close or destroy the data object.
- A client checks the returned exception status.
- Upon success, the client gets returned values from the same or new data object.
- The data object may be destroyed or stored persistently.
- The client can close and destroy the connection to the object implementation.

While CORBA clients and object implementations can use the OMG IDL repository to “discover” new services and learn about their interfaces, the object implementation metadata exchange provides much more detail as to the type of data that clients can exchange with the object implementation

and what data the object implementation currently has in its databases. The object implementation also can return sample scripts that clients can execute to retrieve this data.

The following types of exchanges are legal.

- **DS\_ap\_GETDATA** This exchange type may be used, for example, to ask a clipboard for a data object found in its paste buffer.
- **DS\_ap\_GETMETADATA** This exchange type may be used, for example, by a client to find out about object implementation metadata. A client may ask the trader service for metadata consisting of the trader database schema. A map front-end may ask the map object implementation for metadata describing the map products and overlays it may have available.
- **DS\_ap\_PUTDATA** This exchange type may be used, for example, to send a front-end application an existing object for editing.

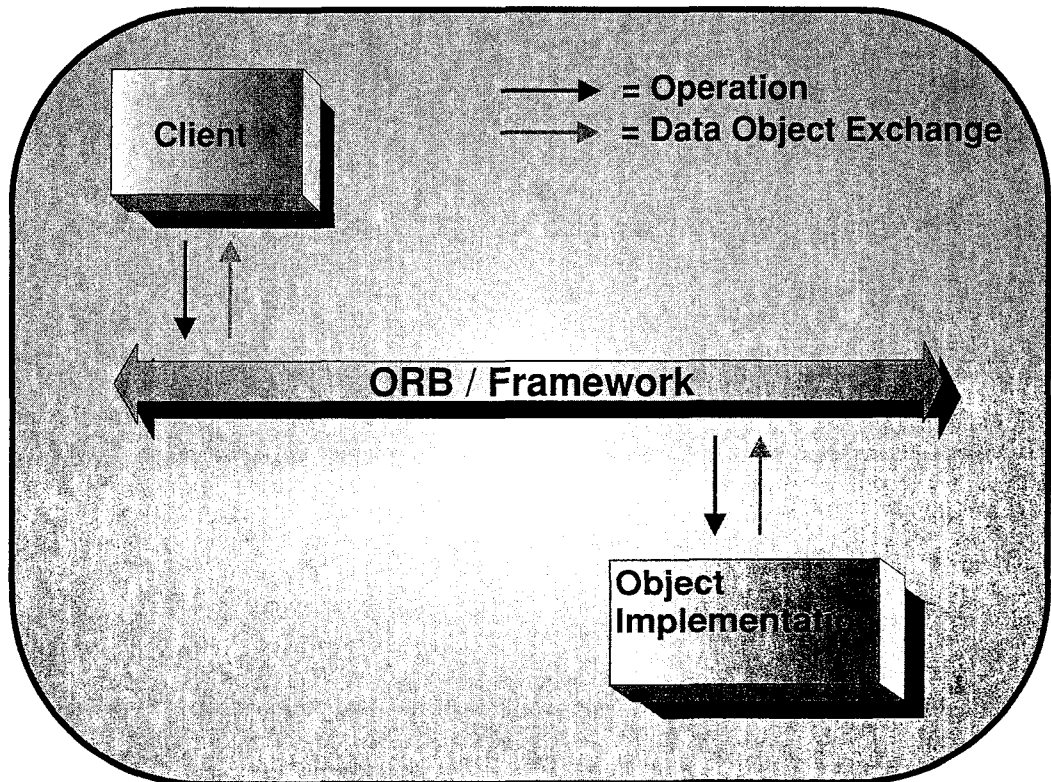


Figure 7.7. GET\_DATA and GETMETADATA exchange.

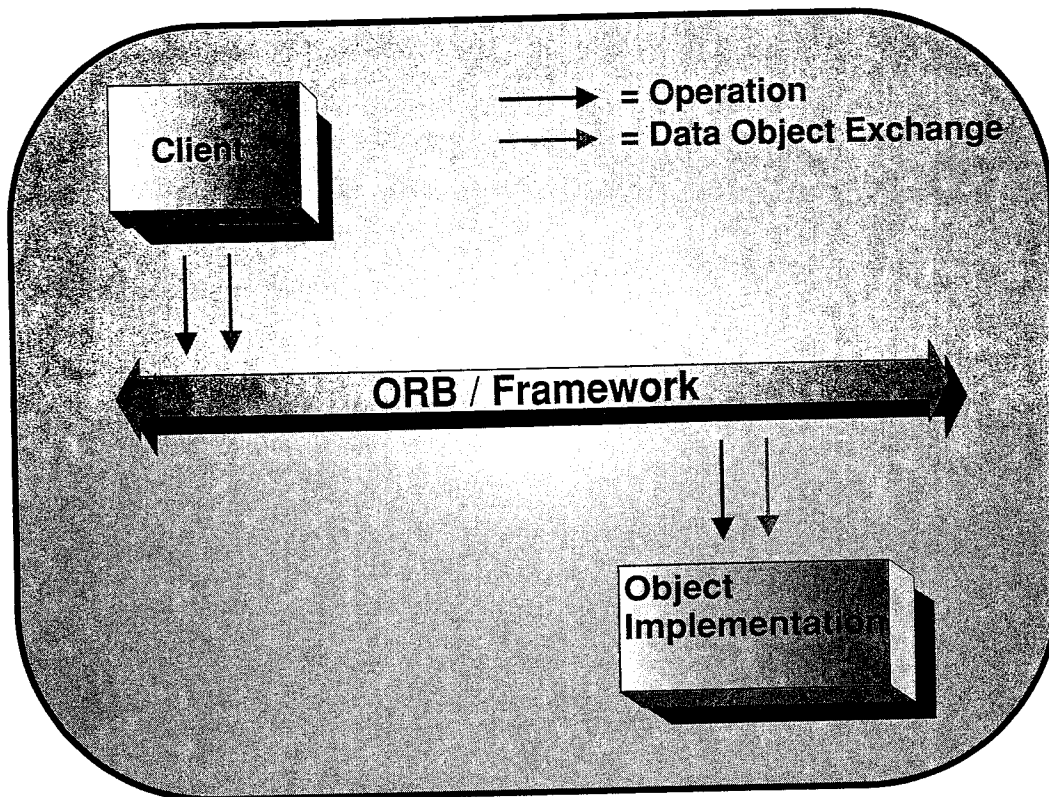


Figure 7.8. PUTDATA exchange.

- **DS\_ap.OPENFRONTEND** This exchange type may be used, for example, to send an initial object to a front-end application. When activating a map front-end to retrieve a map object, the initial exchange with the front-end may be a simple object that may have the requested map width and height but none of the other map properties (which the client may not know about yet).

```
void exchange (      in   Operation exchangetype,
                    inout dt dataobject )
                    raises ( NOT_OPEN,
                              INVALID_EXCHANGE_TYPE )
                    context ( DS_context_attributes );
```

**Query** The query operation is provided to allow for the retrieval of data objects when no information is available ahead of time as to the type of objects or their properties. The query string can be made in any script language



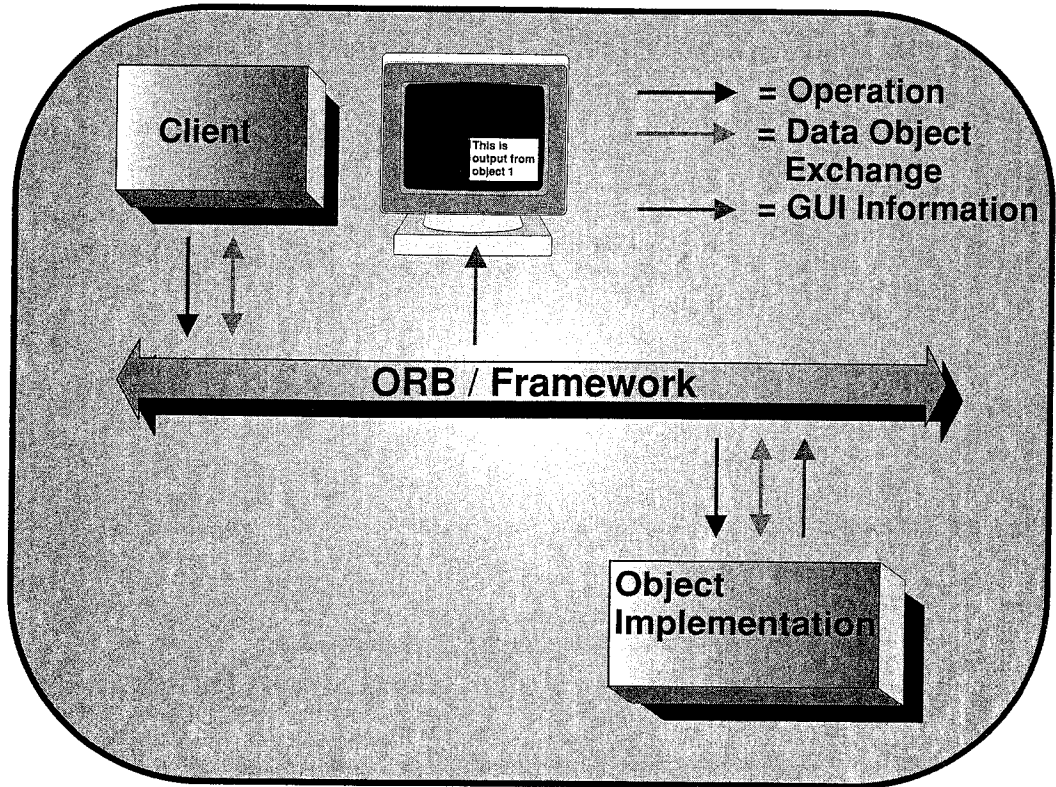


Figure 7.9. OPENFRONTEND exchange.

and may be created by the application or retrieved from the trader or object implementation as metadata. In the last two cases, the client application need not understand the object implementation's script language in order to perform the query. (See the section titled "User Interface Functions" on page 220 for more details).

The Query operation provides a common interface to data sources (Figure 7.10). A dynamic query string parameter is tagged with the identity of the query language. The object implementation returns the results in a data object, which may contain a table object for tabular results. By convention, data source object implementations should provide a `query()` implementation to return data objects and an `exchange()` implementation to return metadata objects.

```
void query (
    in Script query,
    out dt respondedataobject )
raises ( NOT_OPEN, UNKNOWN_OPERAND,
```

```

UNSUPPORTED_QUERY, INCOMPLETE_QUERY,
UNSUPPORTED_LANGUAGE, SCRIPT_SYNTAX )
context ( DS_context_attributes )
    
```

**Convert** The most prevalent problem in integration is related to data formats. Issues include proprietary formats, incompatible formats, different versions of the same format, and loss of data during conversions. The problem DISCUS addressed was simplified due to the ORB ability to convert data at lower levels. For example, integrators no longer have to worry about reverse-byte ordering between different operating and hardware systems.

The convert operation is provided to allow for the conversion between various data formats. The operation is independent from the type of data, so that conversion servers can convert between image formats, map formats, document formats, spreadsheet formats, and so on. Object implementations may choose to implement their own conversion methods (where a vendor supports multiple conversions to its own native format), or they may choose

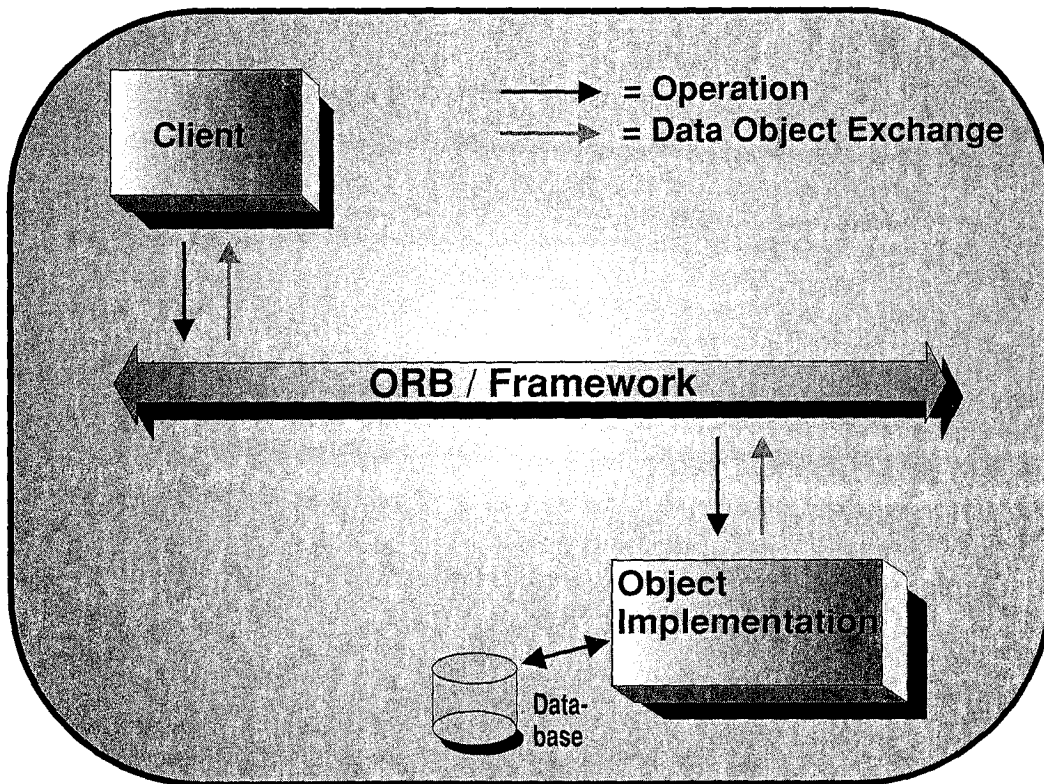


Figure 7.10. Query operation.

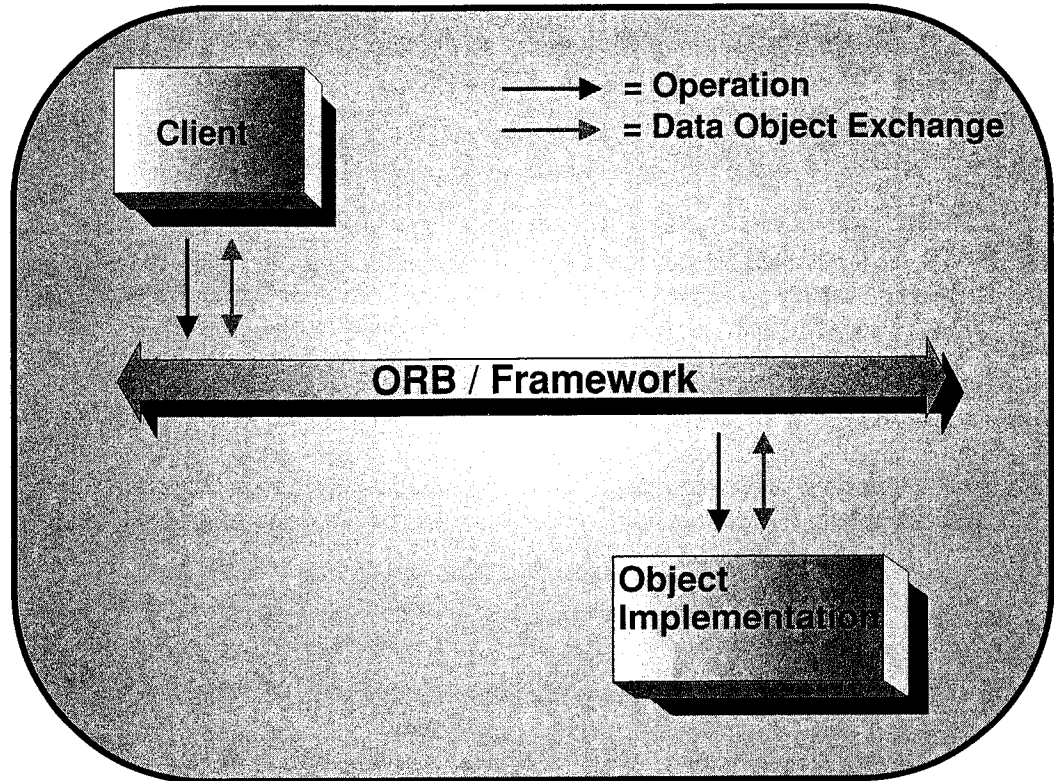


Figure 7.11. Convert operation.

to provide conversions via a conversion server that can convert between several formats.

By convention, if an application introduces a new data format, then it must provide a `convert()` service that allows other applications to convert the data to a commonplace format. The new service is registered with the trader to advertise its existence. All DISCUS applications should make use of the convert operation because of the large number of alternative and competing standards/nonstandard data formats available today. Usually each application has some notion of its “preferred” formats. Upon accessing a data object and a property, applications can check if the format of that property is not the one they expect. In these cases, the convert operation could be invoked automatically. Users may experience an additional slight delay; however, they are no longer required to use import or export menu options, or limited to the set of conversion filters offered by the particular application.

Upon return, if the convert is successful, the data object reference points to the same object with the modified property.

```
void convert (
    in string format, // desired data format representation
    in string propertyname, // property containing input data
    inout dt dataobject ) // object containing formatted data
raises ( NOT_OPEN, INPUT_FORMAT_UNKNOWN,
        OUTPUT_FORMAT_UNKNOWN )
context ( DS_context_attributes );
```

**Execute** The execute operation provides a way for applications to communicate with other applications via some scripted language. The operation allows the specification of statements for operation as well as a list of input data objects to be operated upon. The operation may return one or more data objects as a result of the script. The execute operation may be used, for example, to send two input image objects to a registration algorithm and receive the output single image object as the output data object.

By convention, an object implementation's script language should provide access to functionality, by including operations accessible from legacy

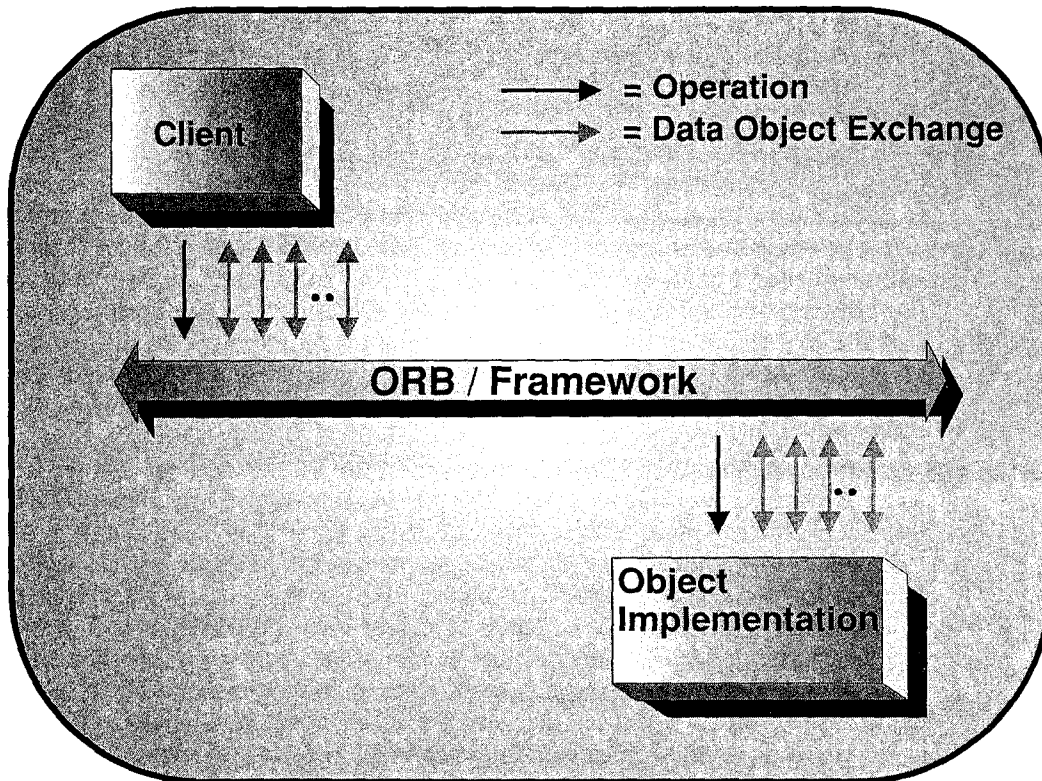


Figure 7.12. Execute operation.

API's and any user interfaces. Object implementation metadata can contain some canned scripts. The execute operation can be used to produce combinations of other operations, such as the exchange and query. However, because excess flexibility often results in less interoperability, Execute should be used when the simpler and more direct exchange and query cannot support the desired functionality.

```
void execute ( in   Script      commandlist,
              in   SeqObject   inputdataobjects,
              out  SeqObject   outputdataobjects )
  raises ( NOT_OPEN, UNSUPPORTED_LANGUAGE,
          SCRIPT_SYNTAX, UNKNOWN_OPERAND )
  context ( DS_context_attributes );
```

## DATA AND TABLE OBJECTS

### Data Objects

The data object class private attributes are sets of name-value pairs. We picked this approach in order to provide a simple conceptual approach to data objects and a simple implementation for developers. The name-value pairs are nothing more than a flat single list of CORBA Type "any". The data object is therefore nothing more complex than a CORBA NVList, and a factory can be as simple as relating NVLists to opaque data object references. Because of this approach, we were able to define a rather simple method of set/get operations to store or retrieve properties. Using CORBA Types, however, allows each property to be as complex as required. Even though the property list is flat, a property may be a sequence of structures that can contain an integer, an array, and other sequence members.

Data objects are simple containers for related data. A data object is a collection of named values; new named properties can be created, and the property may be a value of any type (passed as CORBA type "any").

For interoperability, DISCUS defines additional conventions. The framework defines an enumerated set of types of data objects, such as TEXT, IMAGE, GIS, and so on. When a data object is created (create\_dat()), an enumerated type parameter identifies its type and the factory initializes the appropriate name-value fields. An object hierarchy is defined for the data object name-values. This hierarchy is separate from the DISCUS framework IDL, so that this list can be extended without changing any API's. The root class of the name-value hierarchy is a set of name-values shared by all data objects (the Common object). Specialized application classes, such as GIS, share common name-values as defined by the hierarchy (Figure 7.13).

By convention, each application supports all of the named-values defined in the hierarchy for its application class. Applications may add additional name-values, which other applications can disregard.

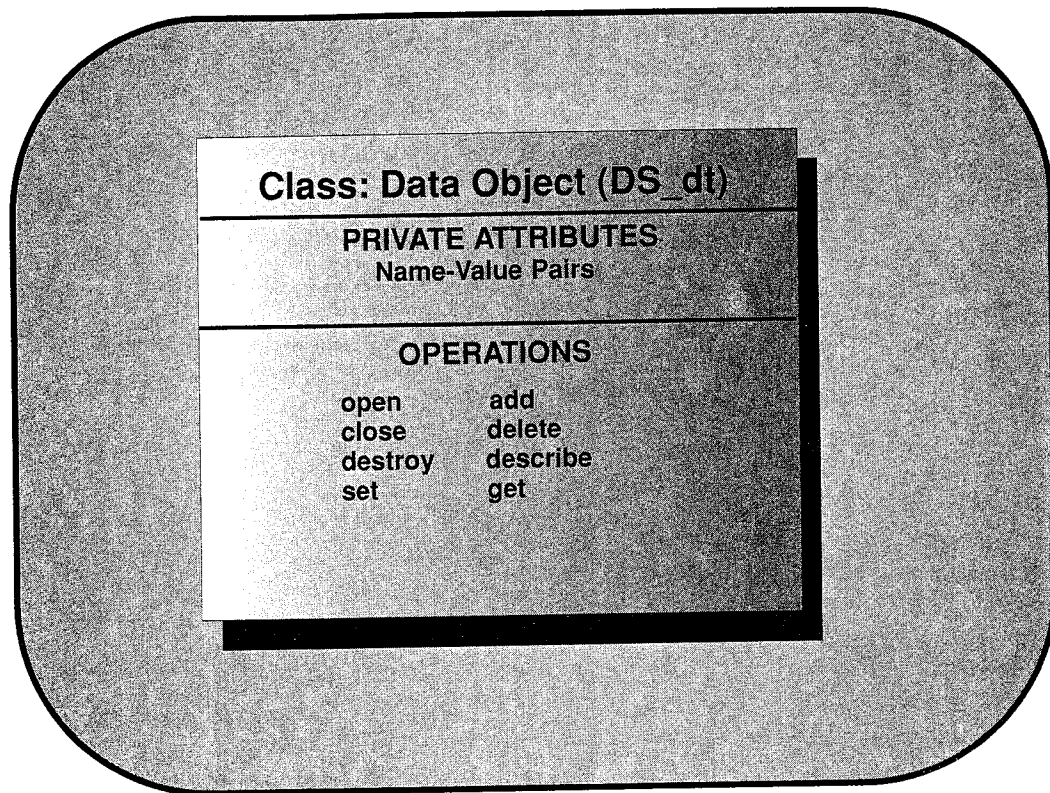


Figure 7.13. Data object class definition.

The protocol for creating and accessing data objects includes the following sequences of operations:

```

create_dat()           // create the data object
open()                // open the data object
set()....             // one or more sets

// optional add() to create new named values
// optional delete() to eliminate named values

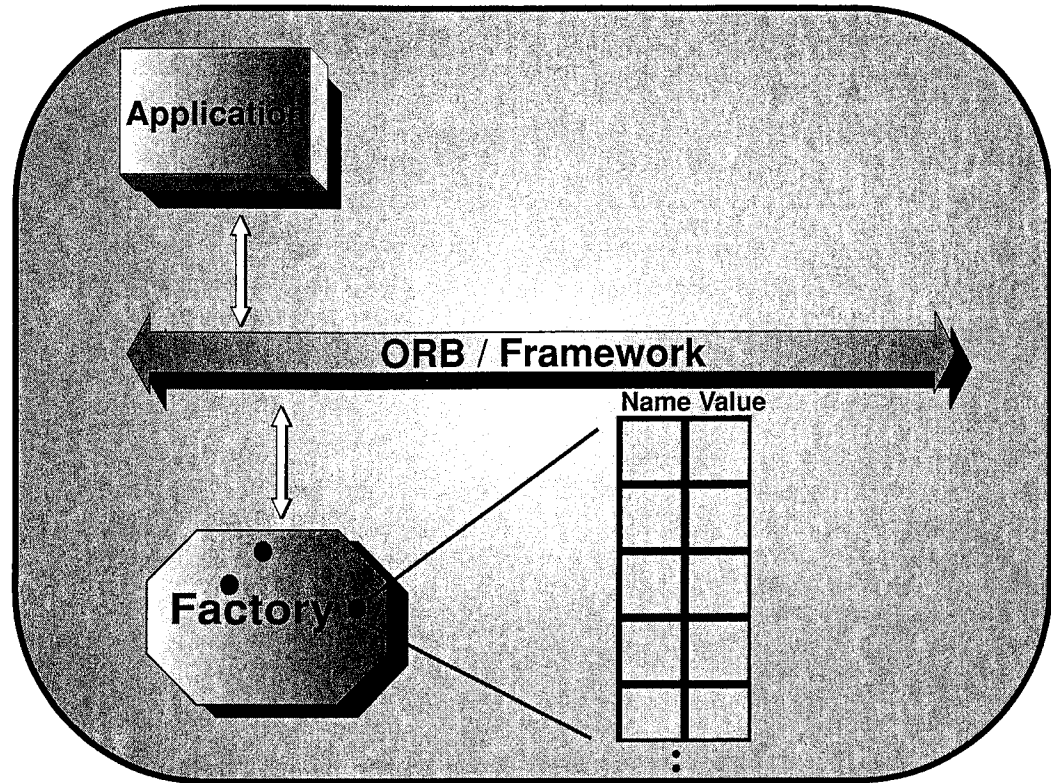
close()               // close the data object

// share with other applications using exchange(), convert(), query(),
// or execute()
// these perform an implicit close ,OR

destroy()

```





**Figure 7.14.** A data object is made of Name-Value pairs of properties.

A typical sequence of operations for an existing data object may be:

```

open()                // open the data object
get()....            // one or more gets

// optional usage of add(), delete(), or set()

close()              // close the data object

// optionally: share with other applications , OR

destroy()

```

Data objects are accessible to clients and object implementations only via an opaque object reference. Once the data object is created, it must be opened in order for an application to have access to its properties. When a

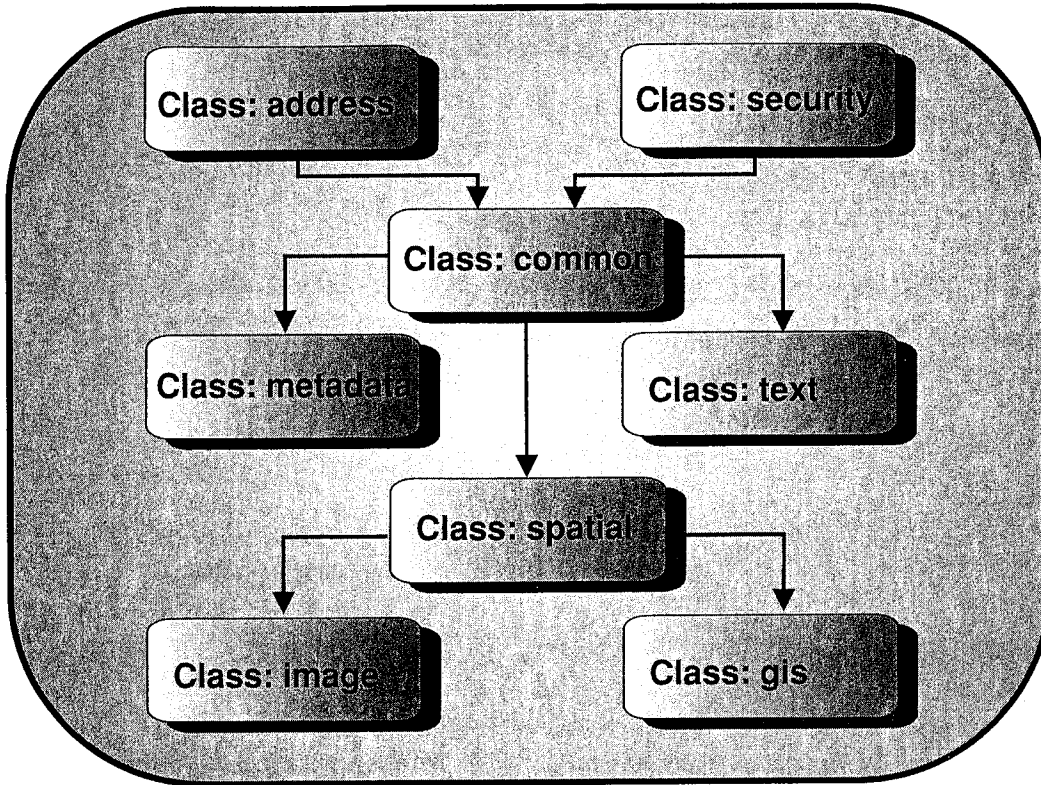


Figure 7.15. Data object hierarchy.

data object is exchanged between applications, the sending application no longer has access to the data object properties. The application cannot access the properties when the data object is destroyed or closed.

## OMG IDL SPECIFICATIONS FOR DATA OBJECTS

The data object interface inherits from the common object. It defines exceptions to indicate that the object was not found, that the object already exists, that a property was not found, or that a TypeCode was not found. Data objects property lists can contain any CORBA types, including user-defined types. Properties can contain other data object references, thereby embedding data objects within data objects.

```

// DATA OBJECT INTERFACE
interface dt: DS::CommonObject {

```



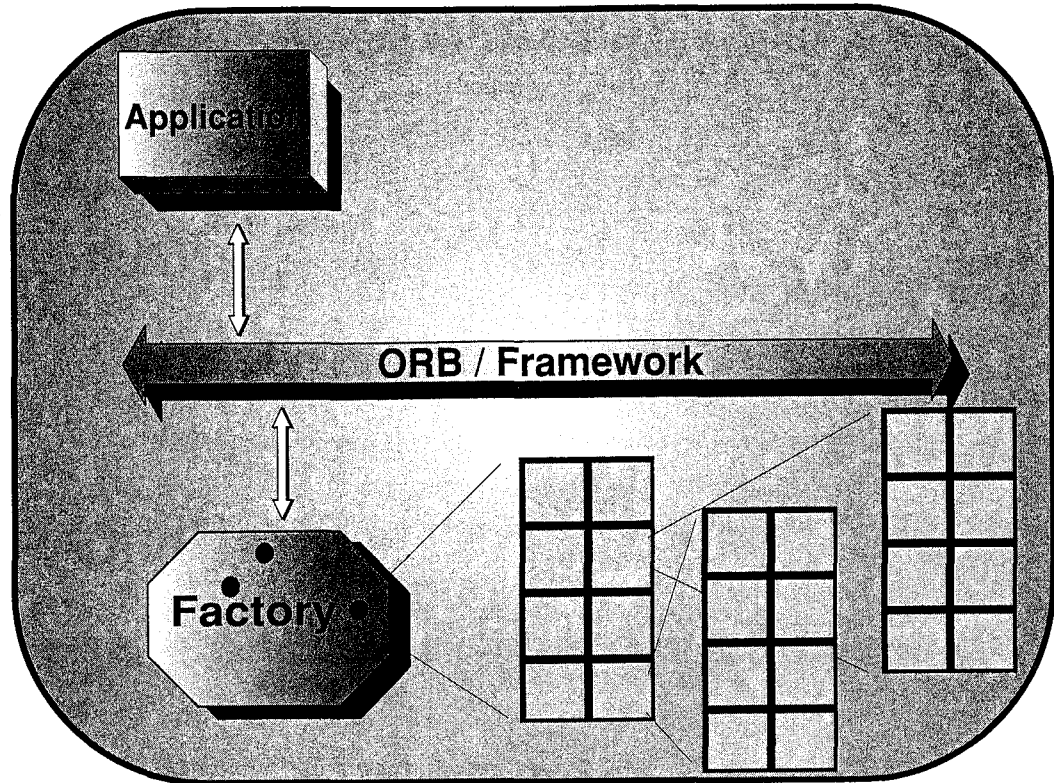


Figure 7.16. Data objects can contain other data objects.

```

readonly attribute string objectType;
//Structure useful for storing data object properties with
//formatted values.
struct FormattedDataRep{
    string format;
    any value;
};

//Add a property to a data object
exception ALREADY_EXISTS DS_exception_body;
void add ( in string newpropertyname )
    raises ( NOT_OPEN, ALREADY_EXISTS )
    context ( DS_context_attributes );
//Delete a property from a data object
exception NOT_FOUND DS_exception_body;
void delete ( in string newpropertyname )

```

```

        raises ( NOT_OPEN, NOT_FOUND )
        context ( DS_context_attributes );

//Retrieve the names and types of a data object
typedef sequence<string> NameList;
typedef sequence<TypeCode> TypeList;
void describe ( out unsigned long numberofproperties,
                out NameList names,
                out TypeList types )
                raises ( NOT_OPEN )
                context ( DS_context_attributes );

//Set the value of a data object property
exception TYPE_NOT_FOUND DS_exception_body;
exception PROPERTY_NOT_FOUND DS_exception_body;
void set ( in string propertyname,
           in any value )
           raises ( NOT_OPEN, NOT_FOUND,
                  PROPERTY_NOT_FOUND )
           context ( DS_context_attributes );

//Get the value of a data object property
void get ( in string propertyname,
           out any value )
           raises ( NOT_OPEN, NOT_FOUND,
                  PROPERTY_NOT_FOUND )
           context ( DS_context_attributes );

}; /* end interface dt */

```

**Open** The data object interface open is inherited from the common object. The reference of the object to be opened is the reference of the data object created by the factory. If the data object exists and is accessible by the application, the operation returns successfully.

Only one application object may have the data object open at any time.

**Close** The interaction between the open, close, and destroy is very important. The close indicates that the application no longer requires access to the data object. A data object must be closed before it can be shared with another application through framework operations such as exchange(), convert(), query(), and exchange(); otherwise an exception is returned. The section titled “Managing Data and Table Objects” on page 205 discusses how the factory affects this interaction.

**Destroy** The data object must be opened by the application in order to destroy it. Once the data object is destroyed, the reference and data associated

with the object is removed. In the linked-in implementation of the framework factory, the reference count is not maintained because copies of data objects are exchanged. Data object maintenance is completely implementation dependent and can be as complex as needed. The factory or the object database can keep count of references so that when linking is available, all applications that point to the object can be notified, or the destroy will be denied until the reference count is zero.

**Set** The set allows the modification of a property of a data object. In order to simplify memory management, the set is destructive and copies all data. It is the responsibility of the factory or data object implementation to remove existing data, and it is the responsibility of the application to remove its copy of the data.

```
void set ( in string propertyname,
          in any value )
          raises (NOT_OPEN, NOT_FOUND,
                PROPERTY_NOT_FOUND )
          context ( DS_context_attributes );
```

The set accepts a property in the form of a CORBA type “any” so that it is self-describing. For example, for a type “any” of type “string”, the value would point to a string. For a type “any” of type “integer”, the value would point to an integer. For a sequence, the sequence TypeCode within the “any” describes the elements of the sequence and its length.

**Get** The get operation retrieves the data and type associated with a particular property. A copy of the data is given so that there is complete separation of the data object and application data. The application is responsible for removing any data it receives. The get operation implementation could be dependent on context variables that can signal whether a copy of a pointer should be passed in the case of very large data (or file pathnames).

```
void get ( in string propertyname,
          out any value )
          raises (NOT_OPEN, NOT_FOUND,
                PROPERTY_NOT_FOUND )
          context ( DS_context_attributes );
```

**Add** The add operation allows for the dynamic addition of data object properties of any type. While this feature is very powerful, its use should be limited. Applications should expect other applications to recognize only agreed upon properties for a given type of data object. It is possible to describe the contents of a data object dynamically. (See the section titled “Describe” on page 195.) However, code to interpret previously unknown properties may be

complex. To avoid the complexity, DISCUS application code should be written to manipulate known properties of known types. Each application type has a set of specific properties (for text objects, map objects, etc.), and there is a hierarchy of properties defining the more generic common properties.

The addition of new properties is a framework capability that can be used for extensibility.

```
void add ( in string newpropertyname )
         raises ( NOT_OPEN, ALREADY_EXISTS )
         context ( DS_context_attributes );
```

**Delete** The delete operation allows for the elimination of a property from a data object. The property must have been added using the Add operation. An exception may be returned if the property is standard for the type of data object.

```
void delete ( in string newpropertyname )
            raises ( NOT_OPEN, NOT_FOUND )
            context ( DS_context_attributes );
```

**Describe** The describe operation allows an application to get a description of the metadata that comprises a data object. The operation returns the number of properties, a sequence of strings that contain the property names, and a sequence of TypeCodes that describe each property type. The information may be used to interpret the data automatically, or to display the contents of the data object to a user within some graphical-user interface (GUI) and allow the user to peruse the information or to act on it (for example, display the image contained in a Display\_Rep property using some imaging display tool).

```
void describe ( out unsigned long numberofproperties,
               out NameList names,
               out TypeList types )
              raises ( NOT_OPEN )
              context ( DS_context_attributes );
```

### Data Object Properties

Data object properties predate the OMG standards process for properties. Data object properties serve a different purpose than the properties defined by the OMG Properties Object Service.

The DISCUS framework supports a set of predefined properties. The framework is designed to be flexible and extensible, and we expect that the list of object properties will evolve over time and comprise industry, government, and de facto standards.

The DISCUS framework defines a strawman set of data object property lists, including the basic object, the text object, the spatial object, the image object, the Mapping object, and the metadata object.

These categories relate the data object type to the application type:

OBJ (Common)  
 IMAGE  
 GIS  
 SERVER\_METADATA  
 TEXT

Future extensions:

EMAIL	Textual e-mail, attachments, active mail
DOCUMENT	Editable text with other datatype inserts
PAINT	Editable raster bitmap
OBJ_GRAPHIC	Editable object-oriented graphics viewgraphs
SPREADSHEET	Small-scale editable tabular data
DATABASE	Large-scale data—relational, browsers, etc.
VISUALIZATION	3-D data, scientific, CAD/CAM, sensor simulation
HYPERMEDIA	Linked document collections
MULTIMEDIA	Incorporating real-time media: audio, video

### Table Objects

Table objects are simple containers for tabular data. By convention, a table object is referenced by exactly one data object. The table object stores a matrix of values, passed as CORBA type “any”. In practice, most tables will have consistent types for each column. Table objects should never be passed as data objects to other applications.

The protocol for creating and accessing table objects includes the following sequences of operations:

```

create_tab()          // create a table object given a list of names
                      // and types, cells are empty
open()                // open the table object
set()....            // one or more times

// insert table's object reference in a data object using data object
// set operation

// share with other applications using exchange(), convert(), query(),
// or execute() ,
// these perform an implicit close, OR

destroy()             // destroys the table object and cells

```

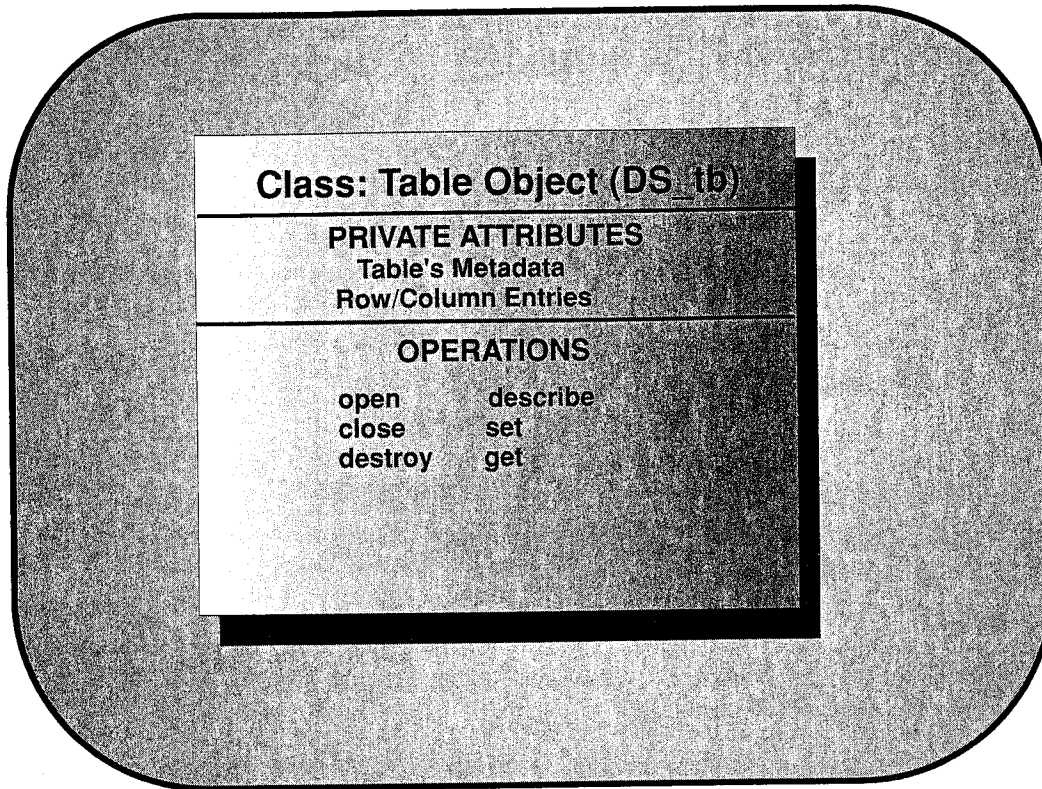


Figure 7.17. Table object class definition.

A typical sequence of operations for an existing table object follows.

```

// get table object reference using data object get operation

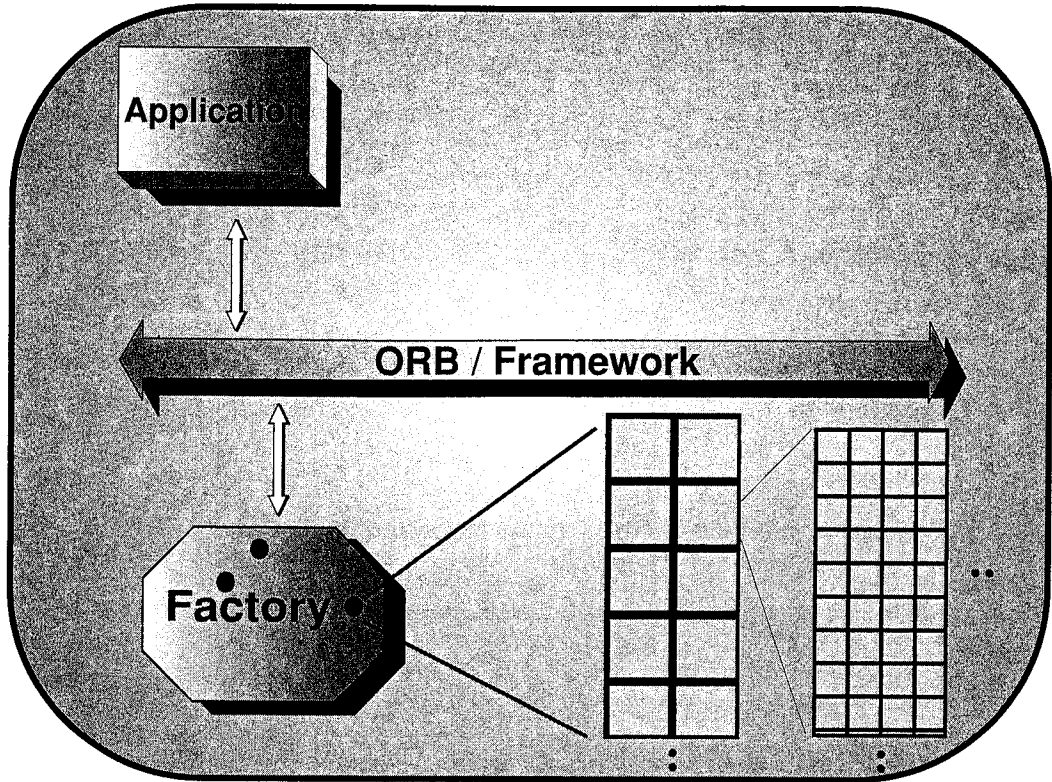
open()           // open the table object
get()....       // one or more times

// optional usage of set()
// optionally share with other applications through a data object, OR

destroy()       // destroys the table object and cells
  
```

Table objects are meant to be a part of data objects and not used as stand-alone objects. They are embedded within data objects using the object reference TypeCode. When a data object containing a table object is destroyed, the table objects it points to should also be destroyed.

Table objects rows and columns can contain any data. This structure makes table objects especially useful to exchange information about appli-



**Figure 7.18.** A Table object is made of row-column cells and is part of a data object.

cations, not only spreadsheet-type data. For example, tables can be used to describe the schema of a database.

```
interface tb: DS::CommonObject {
    exception COLUMN_OUT_OF_RANGE DS_exception_body;
    exception ROW_OUT_OF_RANGE DS_exception_body;

    typedef sequence<string> ColList;
    typedef sequence<TypeCode> TypeList;

    // Retrieve the column names and typecodes of the specified table
    void describe ( out unsigned long numberofrows,
                  out unsigned long numberofcols,
                  out ColList columnnames,
                  out TypeList types )
```

```

        raises ( NOT_OPEN )
        context ( DS_context_attributes );

//Set a value in the table
void set ( in unsigned long row,
          in unsigned long column,
          in any value )
        raises ( NOT_OPEN, ROW_OUT_OF_RANGE, COLUMN_OUT_OF_RANGE )
        context ( DS_context_attributes );

//Get a value in the table
void get ( in unsigned long row,
          in unsigned long column,
          out any value )
        raises ( NOT_OPEN, ROW_OUT_OF_RANGE, COLUMN_OUT_OF_RANGE )
        context ( DS_context_attributes );
}; /* end interface tb */

```

Like the data object, the table object inherits from the common object and also can set some exceptions. In addition to the common exceptions that we may expect to find, such as `ALREADY_EXISTS` or `NOT_FOUND`, there is also `OUT_OF_RANGE`. This is used to signal that an application is attempting to access a data cell that is out of the bounds of the table.

**Open** The open operation allows an application to open a table object. The table object must be opened before it can be accessed. In order to obtain the table object reference, the application first must perform a `DS_dt_get()` on the data object property of type object reference that contains the reference of the table object. Once the reference is retrieved, the application can open the table object.

Only one application object may have the table object open at one time. If an application attempts to open an already open object, it will receive the `ex_DS_tb_ALREADY_OPEN` exception.

**Close** The close operation signals that the access to the table object is complete. A table object must be closed before it can be shared with another application through framework operations such as `exchange()`, `convert()`, `query()`, and `exchange()`; otherwise an exception is returned.

**Destroy** The destroy operation removes all resources and the reference associated with the table object. All data associated with the table object cells is removed. The destruction is shallow, meaning that references to other objects are removed, but the objects themselves are not destroyed.



**Set** The set operation allows an application to set a particular cell of a table by specifying the row and column index. The set is destructive and the table must have been created previously using the factory create operation. The value used to set the cell is of type “any” and must be primed with a TypeCode and value. Because of the use of the type “any”, the table can contain simple or very complex data.

The set operation is destructive, the factory should remove automatically data that may already exist in the cell (or allocate the space for new sets). Applications are responsible for removing their own copy of the data.

```
void set ( in unsigned long row,
          in unsigned long column,
          in any value )
    raises ( NOT_OPEN, ROW_OUT_OF_RANGE,
           COLUMN_OUT_OF_RANGE )
    context ( DS_context_attributes );
```

**Get** The get operation retrieves a copy of the data of a particular cell. The value that is returned is of type “any” and contains the TypeCode and value of the cell. The application is responsible for removing the memory associated with the copy of the data it receives. Applications can look at the column type or the “any” \_type member to determine how the “any” value member should be interpreted.

```
void get ( in unsigned long row,
          in unsigned long column,
          out any value )
    raises ( NOT_OPEN, ROW_OUT_OF_RANGE,
           COLUMN_OUT_OF_RANGE )
    context ( DS_context_attributes );
```

**Describe** The describe operation allows an application to receive information that describes the table. The information that is returned is the number of rows and columns of the table, the column names, and their respective types. The application can use the information to access the table dynamically, and/or to communicate with a user using a GUI (as appropriate).

```
void describe ( out unsigned long numberofrows,
               out unsigned long numberofcols,
               out ColList columnnames,
               out TypeList types )
    raises ( NOT_OPEN )
    context ( DS_context_attributes );
```

**FACTORY OBJECT**

**Factory Concept**

The DISCUS factory is an implementation of the life-cycle service factory concept. The factory is opened and closed by applications to gain access to services that create, destroy, and manage data and table objects. The private attributes of the class include the lists of references of managed data and table objects. The common operations that the class supports are those to open, close, and destroy the factory object. Four additional operations are provided to create and copy data and table objects. The factory inherits from the application class and therefore can support the four standard framework operations. These operations could be left unimplemented, but they can provide valuable functionality. For example, a client may want to use the query operation on the factory to receive a list or status of some data objects. The factory also can provide useful conversions.

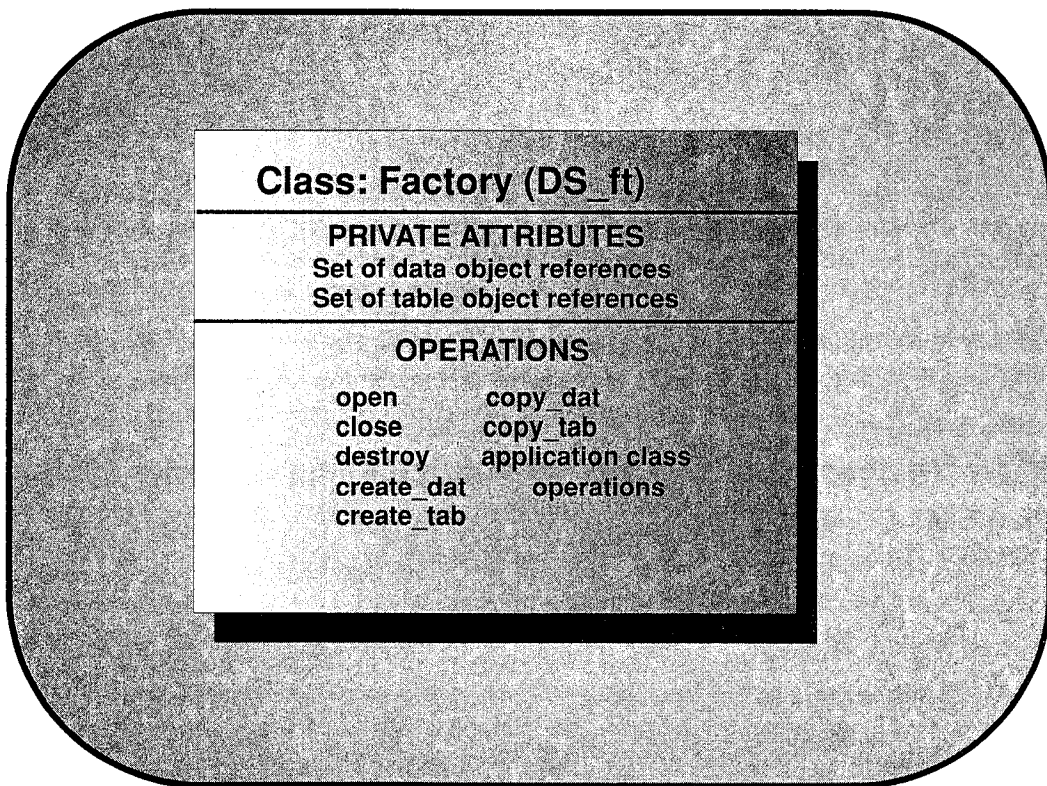


Figure 7.19. Factory class definition.

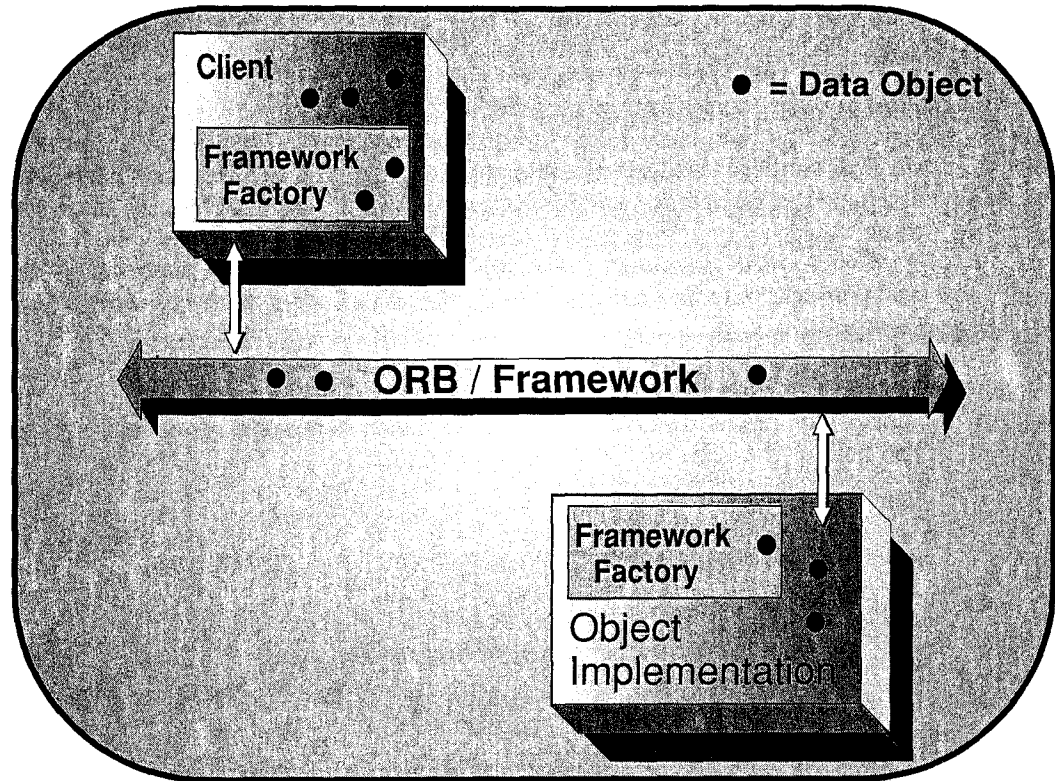


Figure 7.20. Linked-in framework and factory implementation.

The factory can be implemented in various ways.

One way is to have a separate factory object server. The data objects can be external to applications and contained within the factory. Therefore, communications between applications and data objects occurs via the ORB. Another approach is with the factory as a linked-in library. A third possibility is to have each data object implemented as a server. With this approach, a mechanism must be provided to notify the factory that a data object has been destroyed. The linked-in approach has some drawbacks, such as the need to provide data object services to applications across the ORB (other than the application with the linked-in factory). An early DISCUS implementation used a set of collaborating linked-in factories, which could migrate data objects.

The linked-in factory approach provides performance advantages for the local application. Separate factory objects, although suffering from more ORB calls, are scalable and flexible. If performance or resource management becomes an issue, it is possible to create factory objects for each system or even for each user in order to distribute the load.

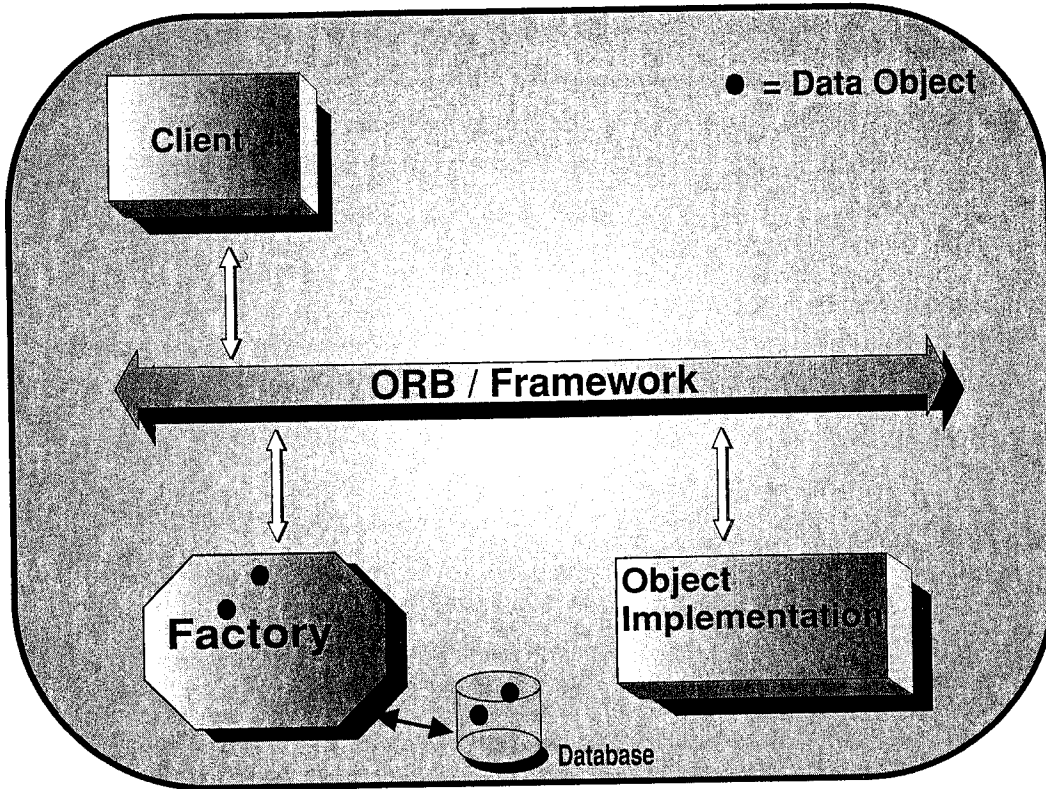


Figure 7.21. Separate factory object.

### OMG IDL Specifications for Factory Objects

The OMG Common Object Services Specifications defines the concept of a factory implementation to create and manage objects [OMG, 94a]. A client using the factory receives only the object reference back from the factory; however, the factory itself has knowledge of the object internals and performs operations, such as initialization.

```
// 0-----0
// |           -- interface ft --
// |
// | The DISCUS factory Object. This factory creates:
// |   *DISCUS data objects (dt:CommonObject)
// |   *DISCUS table objects (tb:CommonObject)
// |-----0
// 0-----0
interface ft: DS::ap {
```

```

// Create a DS::dt DISCUS data object
void create_dat (
    in      string  type,
    out     dt      dataobjecthandle )
    context ( DS_context_attributes );

// Create a DS::tb DISCUS table object
typedef sequence<string> CollList;
typedef sequence<TypeCode> TypeList;
void create_tab (
    in long          maxrows, // maximum number of rows
    in long          maxcols, // maximum number of columns
    in CollList      columnnames, // metadata
    in TypeList      types, // types of columns
    out tb          handle )
    context ( DS_context_attributes );

// Make a shallow copy of a DS::dt DISCUS data object
void copy_dat (
    in dt          dataObject, // DISCUS data object to Copy
    out dt         dataObjectCopy )// The new data object
    context ( DS_context_attributes );

// Make a shallow copy of a DS::tb DISCUS table object
void copy_tab (
    in tb          tableObject, // Table Object to Copy
    out tb         tableCopy )// The new table object
    context ( DS_context_attributes );

}; /* end interface ft */

```

**Open** The factory inherits the open operation from the application class. There can be more than one factory object. DISCUS data objects are extensible, so that the factory can support additional data object properties without having to change. If other factories are introduced, it should be possible to provide a simple gateway or conversion service to convert from one factory's data objects to another's.

The DISCUS framework factory can be called in C.

```
status = DS_ft_open(factory, &Ev, &Ctx);
```

The open creates the required resources to allow clients and object implementations to create, manage, destroy, and access data and table objects.

The call initializes the internal object tables and framework resources. In the future the factory could be used to initialize various frameworks, or many factories may be available.

The framework implementation minimizes how much CORBA the programmers must know and performs standard context and object implementation registration calls whenever possible. It provides a simple interface to the technology. Nothing prevents, however, developers from using any standard CORBA calls to interface with the ORB directly.

**Close** The factory close is also inherited from the application class. The object reference has to be the same one as an associated open factory object. After a close, the application cannot access any of the existing data objects.

Prior to terminating execution, the clients and object implementations should close the factory in order to release some of the framework resources. The close signals a termination of the current session.

```
status = DS_ft_close (factory,
                    &Ev,
                    &Ctx);
```

**Destroy** The factory destroy also is inherited from the application class. The object reference has to be the same one as an associated open factory object. The destroy signals the factory to remove all resources (e.g., cache storage) and data/table objects associated with the current session that an application has with the factory object. Unless specifically destroyed, objects that are stored persistently are not removed and are still accessible by the application. Depending on the implementation, this may mean that the application has the data object, or only the reference of the object, which may be stored persistently by some database.

### Managing Data and Table Objects

The factory class defines four operations for the creation and copying of data and table objects. Due to the OMG IDL inheritance, each of the data and table interfaces has its own open, close, and destroy so that these operations are handled by the data and table objects themselves. The factory simply returns an opaque reference to these objects; the application can use this reference for any further access.

The `create_dat` operation allows an application to ask the factory to create a data object with a particular set of properties. The factory returns to the application a reference for the newly created data object, and the properties are initialized. See Section Data and Table Objects for a description of data objects.

```
void create_dat (
    in      string type,
    out     dt      dataobjecthandle )
context ( DS_context_attributes );
```

The creation of a table object is similar to that of a data object; however, instead of a predefined list of name-value properties, the table is created dynamically as a matrix of rows and columns. The operation accepts as input a count of the rows and columns, a list of the column names as a sequence of strings, and a list of the column types as a sequence of CORBA TypeCodes. The factory returns to the application a reference of the newly created table object. The table object may then be populated using table object operations or inserted into a data object.

```
typedef sequence<string> CollList;
typedef sequence<TypeCode> TypeList;
void create_tab (
    in long          maxrows, // maximum number of rows
    in long          maxcols, // maximum number of columns
    in CollList      columnnames, // metadata
    in TypeList      types, // types of columns
    out tb          handle )
context ( DS_context_attributes );
```

The copy data object and copy table object operations create new objects that contain the same data. Applications can use these operations to create their own copy of objects. The new objects have new references; however, they contain the same data. This means that if a data object had a reference pointing to another data object, the new data object contains the same reference pointing to the second data object. The copy is therefore “shallow.” Referenced objects are not copied themselves. Applications also can read or write data objects to a persistent store using convenience functions. (See the section titled “Convenience Functions” on page 220.)

```
void copy_dat (
    in dt          dataObject, // DISCUS data object to Copy
    out dt        dataObjectCopy // The new data object
context ( DS_context_attributes );

// Make a shallow copy of a DS::tb DISCUS table object
void copy_tab (
    in tb          tableObject, // Table Object to Copy
    out tb        tableCopy // The new table object
context ( DS_context_attributes );
```

## FRAMEWORK SERVICES

### The Trader Service and Metadata Objects

The trader service and metadata objects enable clients dynamically to discover new applications and get information about services. These facilities complement the metadata in the interface repository and the Common Object Services Specifications (COSS) Naming Service.