4

Distributed computing, middleware and migration

From each according to his abilities, to each according to his needs!

K. Marx (*Critique of the Gotha Programme*)

The theme of this chapter is sharing: the sharing of both data and functions by different systems and across different platforms and processes. This leads us to the consideration of the technology of distributed systems, client-server computing, object request brokers, message-oriented middleware, distributed databases, knowledge management and workflow systems. We begin also to look at component based development (CBD) and enterprise application integration (EAI). Distributed systems are composed of nodes that offer services according to their interface definitions to other nodes as and when required. In this chapter, the emphasis is on the practical issues of migrating from centralized, conventional systems to these distributed systems and we must be aware at the outset that some of the components that are to be distributed will remain conventional while others will be built using object-oriented programming. Above all I will emphasize the rôle of object-oriented models as a way of describing and understanding distributed systems. Many modern companies are determined to base their future computer infrastructure on distributed components as far as possible and must therefore construct a sound migration strategy. If there are mainframes, they will be there for specialized applications with high transaction processing requirements or to act as data servers when an existing corporate database or application cannot be replaced economically. Success in the shift to web-based commerce (e-commerce) will mean that these legacy systems must be integrated with new object-oriented code. Many will be connected *via* XML.

Distributed computing is in some sense a return from the chaos (or freedom) of the PC to the golden (or dark) age of the mainframe. Early computers were single user machines with low storage capacities. Time-sharing operating systems offered the possibility of using the same hardware architecture to support multiple co-operating users. When wide area networks (WANs) arrived, this co-operation could extend across the planet and still only need a few central points of management and integrity control. Workstations, when they arrived took us right back to single user machines where we did our own backups and so on and were once again isolated from other users with whom we might wish to share and co-operate. LANs and the Web ended this quarantine but reintroduced all the complexities of operating a mainframe; only now the 'mainframe' is distributed and harder than ever to manage. Nevertheless, on balance, the advantages seem to outweigh the extra complexity. Distributed computing is, in principle, more resilient, fast, flexible, scaleable and open, and it is needed to support ever more global and distributed businesses.

In this chapter we consider the basic ideas of distributed systems and explain the notions of distributed object computing. I then introduce the ideas of object request brokers and of message-oriented middleware before considering how a migration strategy can be constructed. In Chapter 10 we will re-examine this in the context of e-commerce.

4.1 Distributed and client-server computing

Distributed systems can be thought of as networked computers that do not share memory as do multiprocessors for example. This means that the nodes must communicate by message passing and immediately indicates that object-oriented models may be appropriate for modelling such systems. Also, objects provide a natural metaphor for combining data and control and are the natural units of distribution. Performance is enhanced by the implicit parallelism involved. Expensive, under-utilized resources such as plotters can be shared to reduce costs. In a well designed system there may be no single point of failure or it may be possible to duplicate services so that higher availability and greater resilience becomes possible. Nodes can be added and the system can be reconfigured piecemeal, to reduce costs or facilitate upgrades – rather like component hi-fi systems. These and other advantages must be weighed against the additional overheads of maintaining a complex architecture and the increased difficulty of understanding and describing such complexity. To continue the simile, compact 'hi-fi' systems are much easier to install and use than component hi-fi, though their power, quality and flexibility are vastly lower.

TYPES OF DISTRIBUTED SYSTEM

The operating systems of distributed systems can be **distributed** or **networked** (Tanenbaum and Renesse, 1985). In the former case the operating system is itself

distributed over all the nodes in fragments and the network is largely invisible to the user. Networked operating systems give every node a full copy of its own operating system and the network is visible.

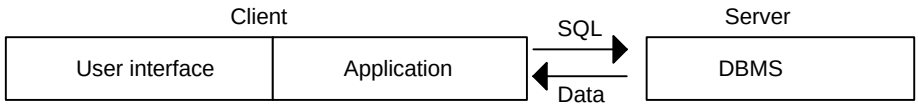
Jazayeri (1992) defines three data management strategies for distributed computing: centralized, replicated and partitioned. **Centralized** management involves placing the data at one node and routing all requests there. The advantage of this is that changes need be made only once. The disadvantage is that all accesses result in a message across the network. **Replicated** management makes copies of the data where they are most often needed. This avoids the need for two-phase commits[1] but may lead to a user working with out-of-date data unless complex additional measures are taken. Where read accesses are more common than writes this is a good strategy. **Partitioned** data management stores data across several nodes, usually based on access demand predictions. This necessitates two-phase commits but can be efficient when the partitioning follows some natural division of data ownership in the business. Object-oriented decomposition is even better because not only ownership but conceptual stability drive the decomposition. Nowadays, database vendors offer a choice between two-phase commits and replication as distribution strategies. Sybase was an early example of a relational, distributed database that offered replication.

Apart from the kind of operating system and the details of data management, there are several different kinds of distributed computer architecture and it is easy to become confused among them. **Client-server** computing involves a single server with one or many clients and most of the intelligence is located in the clients. Often the server is a file-store or a database, although the terminology may be used to include print servers. This is a simple form of distribution, with the GUI and business logic usually entangled. **3-tier** client-server architectures separate the GUI and the business logic, while **n-tier** arrangements make yet finer separations. This is a common architecture for Web-based applications wherein some of the business logic can be run on a so-called **thin** client; i.e. as a Java applet running under a web browser. A further level of complexity is the **multi-client/multi-server** system where there may be several servers but the servers may not communicate directly; in other words: nodes cannot be both clients and servers. Several database products support this architecture. The most general case occurs when every process node may be both a client and a server (in general simultaneously) though this is rare at present. This case involves **peer-to-peer** communication between nodes. Nodes may represent processors or task images within a processor. Messages may be split, relayed and combined as they pass from node to node. Implementing this kind of system is not without its difficulties.

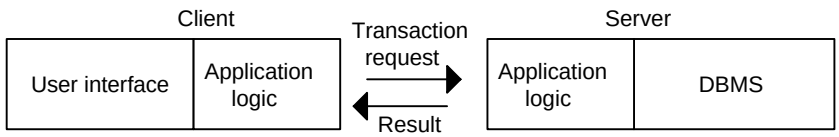
---

[1] Two-phase commit only allows a transaction to be committed on all participating machines when they have **all** signified that they have successfully completed the updates concerned. If any node does not respond 'OK' (or times out) after the first phase of update then all nodes are rolled back to their state before the update.

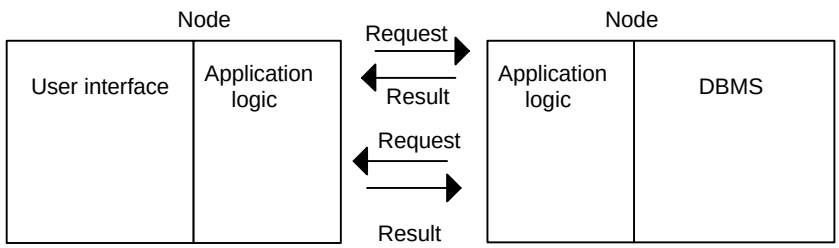
Another way of viewing the different types of distributed architecture is presented in Figure 4.1. Here, we distinguish five models: the database server model, the transaction server model, the peer-to-peer model, the distributed front-end model and the n-tier model. The database server model offers a limited choice as to where to locate the function of the system while the transaction server model is a more effective way to balance the processing load between nodes and reduce network traffic; data intensive tasks can be handled by the server and interface manipulation and secondary computation by the client. Further, SQL queries can be precompiled to improve performance and there is a central point of maintenance.



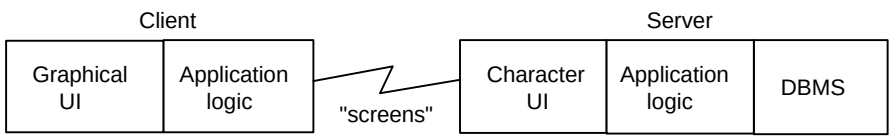
(a) Database server model



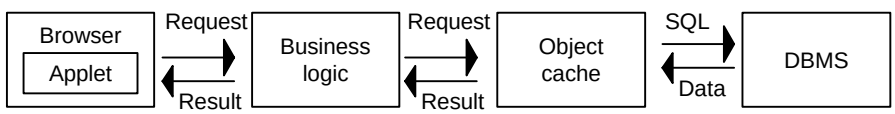
(b) Transaction server model



(c) Peer-to-peer model



(d) Distributed front-end model



(e) N-tier model with thin client

Figure 4.1 Distributed computing architectures

The peer-to-peer model is the most flexible and general but the most difficult to program and manage. In applications such as real-time process control, workflow and groupware it is almost mandatory. The distributed front-end model is really only used as a sort of wrapper strategy when there is a legacy mainframe with a character interface based on synchronous terminal protocols such as 3270. Here, the workstation client should be envisaged as re-mapping between its own and the host screen format and transmitting or receiving the results.

The 3-tier model is increasingly common but we will see later how it is being supplanted by peer-to-peer and n-tier models based on middleware that hides much of the complexity of such models.

CLIENT-SERVER MODELS

Client-server computing (CSC) can be defined as the division of processing and data between one or more front-end client machines that run applications and a single back-end server machine which provides a service to each client. This is often taken to mean that the machines are connected by a network and that the clients are workstations running a graphical user interface. More generally, a client-server system can be defined as one in which some element of the computation, user interface or database access is performed by an independent application, as a service to another; possibly on the same machine. In this broader sense all object-oriented systems are client-server systems, though the converse is not true.

Client-server computing, as we have seen, is a special case of distributed computing but does not necessarily involve a distributed database, though it may. One should avoid the temptation to confuse it with terminal emulation on workstations or with purely graphical front-ends. Further, CSC is not a new invention. In the 1970s terminals were often attached to front-end processors (usually small minicomputers) which switched the services of a mainframe application (often a database or modelling system). By the 1980s we were using PCs for terminal emulation and a small amount of the application logic, often concerned with display, had migrated to the PC. CSC became a mature possibility with the advent of multi-tasking PC operating systems so that the user could maintain links with a remote server while running a local application as well.

It is worth noting that resource sharing systems such as file servers are not in the same category as database servers. With a file server, the client application requests a file which is then locked and transferred to the client for processing. If the access is for update, the file may be locked until the client releases control. All the processing is done by the client. With a database manager running on the server, the client issues queries and updates. The server processes these and returns only the result. Processing is shared, network traffic is reduced and locking is minimized.

Most client-server systems are of the database server type. However, there is a range of types, as depicted in Figure 4.2. The division between client and server

can occur at any horizontal line in the diagram. Application servers correspond exactly to objects and object wrappers. We could slightly generalize the idea of a database server to that of a domain object server and differentiate it from an application object server. This point of view emphasizes a layered architecture rather than a single division between client and server as in the figure. We then begin to see that object-oriented systems are nothing other than, possibly layered, multi-client/multi-server or peer-to-peer systems. Figure 4.4 suggests six layers; the right number depends on the application. The hatched areas show opportunities for the introduction of extra layers.

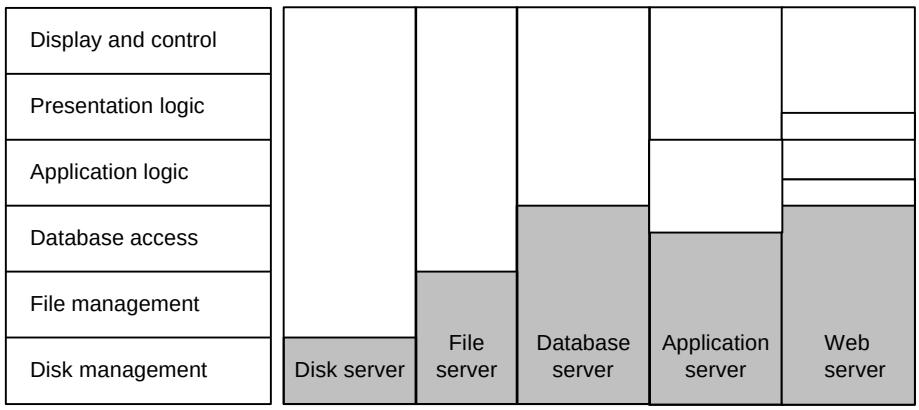


Figure 4.2 Levels of abstraction of client-server systems

Conventional database tools are often organized around the client-server metaphor. One of the earliest commercial products to do this was Sybase, with more recent versions of Ingres, Oracle and other products working as multi-client/multi-server systems. The original Sybase approach offered each client a single access point to a multi-threaded server. There was one process and each user represented a thread of control within it. This is efficient in terms of memory utilization but cannot be exploited by modern multi-processor machines or parallel processors. The multi-server approach offers each client a single-threaded process of its own and is thus far better able to exploit parallelism even through slightly more hungry for memory. These kinds of tool assume network support and adherence to certain standards. The client and the server applications communicate *via* an API (such as the Sybase 'Open Client' API), connectivity software (such as 'SQL Connect') and some network protocol (such as TCP/IP, IBM's LU6.2/APPC or Novell's SPX/IPX). With the relational products, what is transmitted through all these filters is usually SQL, but this need not be so for object-oriented databases or other kinds of application. Again, with relational systems, the ones that make use of stored procedures are able to reduce network traffic considerably. This is because whole transactions are not fragmented into separate statements and they may be

pre-compiled to improve performance further. Object-oriented systems, of course, give these benefits automatically, provided that the methods really are stored with the corresponding objects. Multi-client/multi-server systems extend the CSC model to give users heterogeneous access to several servers without the need to know where the services are located.

Stored procedures in relational systems provide not true object-orientation but the possibility of regarding the database server as one huge object; i.e. of creating a wrapper. The difficulty is that the encapsulation is not enforced by the system and the discipline of a robust approach to object-oriented analysis must be enforced by the standards, procedures and QA policies of the organization. It is quite possible for the undisciplined – or badly educated – developer to access the database structure directly using SQL rather than by going through the access routines supplied as stored procedures. One rather unwieldy solution to this problem is for the database administrator to remove all access privileges from all users so that the only access must be *via* stored procedures; but this also removes most of the advantages of a relational database in terms of being able to perform *ad hoc* associative queries. 2-tier client-server architectures are now regarded as far too inflexible for modern business needs.

CENTRALIZED *VERSUS* DISTRIBUTED

There is a contradiction between the need to have all applications inter-operate with each other and the need to permit each department to work in its own way. Housekeeping too becomes a more complex task. On the one hand, users may not consider routine back-ups and disk purges as part of their job but, on the other, nothing is more annoying than when the Information Systems Stasi comes round and erases all those useful little files on your hard disk over the weekend. One possible solution is automated housekeeping but the danger of a totalitarian approach remains.

There are yet other areas where distributed systems increase complexity. In a centralized system the operators and administrators take backups and tune the system. In a distributed one each user must be both operator and administrator. Physical distribution often makes central backups and software release control impractical. Network shared file servers help but they fail to provide a complete answer. A true distributed system implies a distributed operating system that makes the multitude of systems look like one system as far as both its users and administrators are concerned. Such a system must ensure both fault tolerance and parallelism. That is, no single part failing should bring the system down and there must be interconnected units containing both processor and memory. The first requirement implies that the system nodes must share state or, in the present context, that objects must be replicated.

The sheer cost of a simple solution may also increase complexity. Fully interconnected networks are simple but expensive; and thus the need for complex routers and scheduling techniques. Mullender (1989) illustrates this point well by comparing the operating policy of a railway connecting two towns with a track in each direction with one having a single track, possibly plus sidings for passing.

This example also illustrates the concept of **latency** in networks, which is too often ignored by designers of distributed systems. At 60mph a single track network (with no sidings) connecting two points 120 miles distant has a latency of 2 hours while the double track railway has a latency of almost zero. Latency in a computer network is the time taken to call another thread of control. High latency networks often have severe clock synchronization problems. Latency is an important concept in the design of any object-oriented system, distributed or not. In that context it refers to delays due to blocking sends.

Early adopters of distributed approaches discovered the hard way that existing structured methods not only contribute little but actually impede progress in many areas. This is because the methods of functional decomposition and the separation of data from processing offer no representational techniques for describing communication between autonomous actors (objects). Thus, there is, in these organizations, a profound need for object-oriented analysis methods to help describe the systems being built. Also, new skills are needed; some specifically to do with distributed computing itself, but more significantly in terms of the different kinds of hardware and software platforms that will be used. The rise of XML re-emphasises the need to design interface semantics carefully; despite its power, on its own it is not enough to define semantics.

One of the main problems with distributed computing compared to centralized computing is that the software becomes far more complex, and a strategy is required for managing or reducing this complexity. This is where object technology can help by providing tractable modelling techniques based on message passing, class libraries to protect developers from complex APIs, purpose built routers and full blown object request brokers to simplify location management and inter-process communication even further.

Distributed systems have to be far more complex to prevent problems that just do not arise with conventional systems. If one processor fails while another proceeds, the application may end up in an inconsistent state – leading to the need to be able to roll back portions of an application selectively and intelligently. Imagine what would happen if you posted a cheque to a country whose postal service failed, after you had deleted the money from your account on the assumption that it would be credited to that of the recipient.

There are also problems when two processors access the same data and when the network itself goes awry. Most of us have experienced network errors, which are far more frequent than on stand-alone machines and consequently increased our save rate during tasks such as word processing. In fact, the trade-off between independent communicating nodes and a single network operating system is strongly reminiscent of the trade-off between planned, integrated, co-operative human societies and rampant individualism; each has its advantages – at least for some members.

Most commercial object-oriented databases offer such support, though their architectures vary greatly in terms of how they handle the sharing of storage and computation between the client and server components. The advantage of this

approach is that one can maintain a consistent, fully object-oriented model for the whole application. Also, storage, recovery, transaction management, concurrency and, often, version control are handled automatically by the database. The disadvantage is the potentially large cost of converting existing data and/or applications if the new system is to take advantage of the conceptual model, performance benefits and the possibilities of extending the system using object-oriented programming languages. As Ewald and Roy (1993) point out, where migration in the context of extensive legacy systems is contemplated, the disbenefits can predominate. They remark that existing applications often already have their own methods for instance management, whether as proprietary file management systems or relational database systems. They conclude that object request brokers (ORBs) are often a better solution even though the latter may sacrifice a pure object-oriented style for flexibility. These are flexible precisely because they permit the use of existing applications, packages and other resources within the object-oriented framework. Reuse of such resources, however, is a coarse grain reuse; sometimes violating the principle of small interfaces. Also, an ORB will not dictate exactly where and how applications should be coupled with pre-existing systems.

LOCATIONAL TRANS-PARENCY

Object technology provides a natural way of modelling distributed applications because an object model consists of a set of independent entities, each with its own thread of control, collaborating by message passing; i.e. distributed method invocations. In conventional systems, library interfaces are usually procedural. In distributed systems they are more involved. Just as an object's state is encapsulated, it is easy to see that its location can be 'hidden' in the same way – the location is part of the state. However, some means of finding these objects must be provided. When an object references another (i.e. sends a message) the routing of the message is of no concern to the sender. Providing there is some sort of global address table, the sender should not need to care whether the receiver is even on the same machine. This must be the case for an object-oriented system because objects have unique identity for their lifetimes – regardless of location. In other words the object-oriented model both assumes and implies **locational transparency**.

It is considered axiomatic that distributed computing should offer locational transparency. That is, the user should not need to be aware of the physical location of the services being requested at any time. Even where this is true logically, network delays will sometimes make it only too apparent that requests are being processed remotely. Even finer distinctions become possible when the services are accessed by a mixture of local and wide area networks whose response characteristics are very different. Remote procedure calls (RPCs) violate locational transparency because the client application needs to know where the remote procedure is located to address it. This is sometimes hidden from the user by clever network software such as Sun's NFS and the reusable network services of the OSF's DCE but it remains a problem in principle. The extent to which the services offered by these systems are packaged as objects varies but such packaging, especially of the

API, definitely represents a trend. We will see how distributed object systems overcome this problem in a general way later.

DCE (Distributed Computing Environment) is a software architecture that supports multi-vendor distributed data and process sharing. It protects the user against variations in communications protocols but relies almost entirely on remote procedure calls (RPCs) and is thus rather threatened by the emergence of object request brokers and component models, although some of these use DCE under the covers.

Realizing that locational transparency is possible in principle for object-oriented systems does not lead to the conclusion that implementing such systems is trivial. In fact, a great deal of programming is required to set up the requisite support services and environment, and this involves several technologies as well as networking.

BENEFITS

The benefits most often claimed of distributed computing are that it permits organizations to:

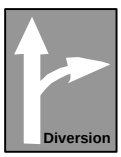
- reduce the incremental cost of meeting the demands of users for increased functionality and access to corporate systems;
- reduce hardware and development costs by allowing the use of tools optimized for particular tasks;
- reduce development times by using existing systems as components;
- reduce hardware costs by utilizing low cost workstations (downsizing) and optimizing the utilization of other hardware (rightsizing);
- increase competitive edge and empower users;
- enable new business models such as e-commerce; and
- ease integration by hiding the complexities of communication between machines and adopting open standards.

Centralized computing offers the advantages of security, economies of scale and easier enforcement of management and accounting disciplines. However, operating costs can be high, systems inflexible, development and maintenance costly and the user has little control – and often limited access. Furthermore, most mainframes are poor calculating machines; so that the users' need to build models of business processes is hard to cater for in this way. This has led to considerable demand for end-user computing and distributed systems but there are still difficulties such as network bottlenecks, failures when not all machines are switched on, costs due to under-utilized CPUs, security breaches, data integrity and file locking delays. New skills are required within many IT organizations and change management is nearly always advisable to ameliorate the level of resistance to change. IT staff need not only their old mainframe skills but skills with PCs, networks and new, complex communication standards and component models. In my view, they will benefit most from an understanding of the principles of OT and, above all, a sound grasp of object-oriented analysis. This too involves, often significant, extra costs in terms of recruitment, consultancy, education and training.

Though distributed architectures usually involve two or more machines, in principle both client and server can reside on the same machine. The distinction is logical as well as physical. Originally, object-oriented programming languages were restricted to only one address space. Objects compiled by different compilers (even in the same language) couldn't communicate with one another, so that class libraries could not be delivered in binary form. Thus technologies such as RPCs, COM and CORBA, arose to tackle the problems of multi-language inter-object messaging. These led on to the emergence of component models and metamodels, including EJB and MOF.

Network nodes can be regarded as abstract data types or objects. However, inheritance and composition links shouldn't cross the network. Associations that span the network should be minimized for reasons of efficiency. Typically, a system layer should be implemented at a single node. In fact, nodes are best regarded as wrapped components.

4.1.1 Network and architectural issues



Distributed systems use more network capacity than centralized ones and it is a common error to underestimate the demand. It is a good idea therefore to involve users – who can often forecast what they want to do, network and operating systems specialists and application builders.

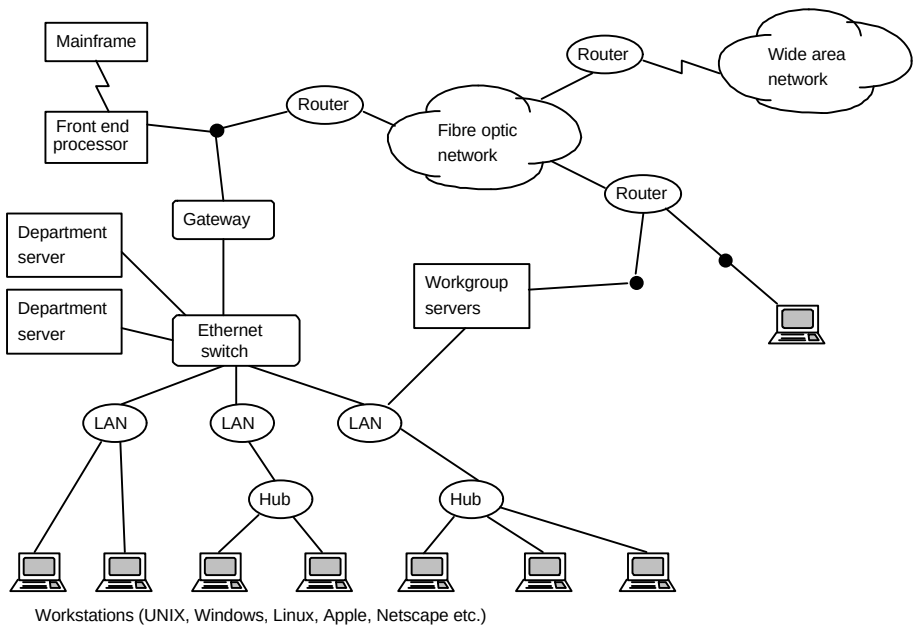


Figure 4.3 A typical network architecture during migration

These three groups together should be able to produce a reasonable forecast of network loading and avoid the situation where people are taking half an hour to move their gigabyte database across the LAN and, incidentally, preventing anyone else from moving data.

Peer to peer communication can be used to build big systems out of relatively small computers. The paradigm for this was DECnet. IBM's APPN protocol and TCP/IP follow the same model, though TCP/IP is more widely supported and less proprietary. It is also the basis for the World Wide Web.

One of the most urgent questions facing an organization migrating to object technology is that of systems, application and network architecture. It is essential that new and legacy systems must be able to interoperate during the migration period. Often the *status quo* is a goulash of different systems, machines and networks. Figure 4.6 shows a simplified version of a typical architecture of the sort required by the migrating organization.

In the figure, the mainframe may be running all sorts of COBOL, FORTRAN, DB2 or VSAM applications and using proprietary network standards, such as SNA, for connectivity. The key component that must be added to the mainframe is some software that allows clients to interact directly with the mainframe relational database, using SQL, or execute COBOL routines remotely. A second key component is the gateway. This is some special software that would normally run on a dedicated machine and which acts as a bridge between UNIX clients and the mainframe network. Several database vendors can supply suitable software. Routers connect different LAN and WAN technologies and hubs permit point-to-point wiring between workstations. Ethernet switches help to maximize both flexibility and throughput. Various other equipment is needed but this is not a book on network architecture and the simplified view presented here will suffice. In future, object request brokers will simplify the picture considerably and it is this development that we consider next.

4.2 Object request brokers and middleware

The need to co-operate transparently with existing systems, packages and other object-oriented systems across telecommunications and local area networks is our next subject. The requirement is to represent applications and services on a network through a common object schema. This schema consisting of objects to represent every service, with all locations and implementations transparent to the user and to other client objects.

Gartner Group define middleware as 'run time system software that directly enables application level interactions among programs in a distributed computing environment' although wags say that it is 'the software that nobody wants to pay for'. In other words middleware is part of the infrastructure of a distributed

computing system. There are several kinds of middleware and several levels of abstraction involved. At the lowest level, message-oriented middleware (MOM), such as IBM's MQ Series or Microsoft's MSMQ, provides asynchronous routing and queuing of messages and guarantees their delivery. MOM provides an intermediate tier in a client-server model. This of course does not ensure that the content of the messages is sensible or, indeed, compatible with the format that the target application expects. For this reason it is common to overlay middleware with a semantic layer, usually based on XML.

One important way in which many packages and applications inter-operate transparently is through the Object Management Group's Common Object Request Broker Architecture (CORBA). Increasingly CORBA is being combined with MOM and Transaction Processing monitors. Current exemplars include BEA Systems' integration of the Tuxedo TP monitor with its ObjectBroker CORBA product and Tibco's ETX.

Founded in 1990, the Object Management Group (OMG) is a large group of influential companies committed to establishing broad agreement between vendors on both the terminology of object-orientation and on interface standards – based on existing technology. Companies originally involved in the OMG included Borland, Microsoft, Hewlett-Packard, Data General, AT&T, Prime, Wang, ICL, Sun, DEC and most of the leading hardware, software and object-oriented suppliers along with several major users such as Boeing and British Airways. At the time of writing there are around 800 members. Meetings of the OMG Technical Committees rotate between Europe, the USA and the Far East, helping to ensure an international base. The OMG is committed to the fast production of published standards, faster anyway than the official standards bodies can operate, and has published *inter alia* an architecture guide and reference model, the CORBA standard for Object Request Brokers (ORBs), a language independent component model, a standard notation for object-oriented analysis and design (UML), specifications for standard business objects in vertical markets (e.g. currencies) and a Meta Object Facility (MOF). Several suppliers offer CORBA compliant products. The fast growth of the OMG suggests that the industry was aware early of both the potential of object technology and the need for standards.

An ORB is a transparent data highway connecting object-oriented applications and object-oriented front ends to existing applications. It is analogous to the X500 electronic mail communications standard wherein a requester can issue a request to another application or node without having to have detailed knowledge of its directory services structure. In this way, the ORB removes much of the need for complex RPCs by providing the mechanisms by which objects make and receive requests and responses transparently. It is intended to provide inter-operability between applications on different machines in heterogeneous distributed environments and to connect multiple object systems seamlessly. It provides a means of using an abstract description of applications and the relationships between them and provides services for locating and using these applications across multi-vendor networks. Applications need not be written in an object-oriented manner

since the ORB effectively provides a wrapper and they can be entire third-party packages. Packages and in-house applications can be reused and combined to deliver brand new cross-platform, distributed business systems. As Ewald and Roy (1993) succinctly put it, ORBs bring the benefit of OT to the world of systems integration.

Object Request Brokers are products based on the Object Management Group's application architecture, illustrated in Figure 4.4, in which objects are classified as Application Objects, Common Facilities and Object Services. Application Objects are specific to particular end-user applications such as wrapped legacy systems, packages or spreadsheets. Common Facilities are objects which are useful in many contexts such as help facilities, compound document interfaces, browsers, E-mail and so on. Object Services provide basic operations for the logical modelling and physical storage of objects and might include persistence (interfaces to databases), naming, event notification, licensing, security services or transaction monitors. The basic idea is that an application that needs to use the services of some object, whether on the same machine or remote from it, should do so *via* a broker rather than by using some sort of remote procedure call that would require it to know the location of the server object. The ORB takes care of locating and activating registered remote servers, marshalling requests and responses, handling

Facilities reuse or extend Services. Domains reuse or extend Services or Facilities. Applications reuse or extend all these and may have custom extensions.

The OMG also defined a standard for interfaces: its Interface Definition Language (IDL). IDL resembles C++ syntactically but may only be used to define interfaces; it is not a full programming language. In the ORB, a request names an operation and its parameters, which may be object names. The ORB arranges the processing of the request by identifying and running a suitable method and returning the result to the requester.

Object request brokers work by acting on requests from all other types of object. They bind these to objects and route requests to other ORBs for binding. They can be regarded either as communication managers or systems integrators since they either route requests to the correct destination object or understand the syntax and semantics of each request by maintaining an object model, or both. They are a small step towards truly intelligent networks. In future, I anticipate expert systems and machine learning techniques being used to take this development even further.

An ORB makes use of the principles of data abstraction using its object model. In this model the **interface classes** specify the services of the applications known to the broker while the implementation classes represent the object wrapper code. To define an interface class one should specify the class in terms of its superclasses, attributes and operations. An Interface Definition Language (IDL) supports both dynamic and static binding to cater for different performance and extensibility requirements. All location services are handled transparently by the broker. This is handled by implementation classes but CORBA does not include a language for specifying these. Vendors are free to innovate in this area.

ORBs extend the polymorphism of normal object-oriented systems based on features such as inheritance to allow the user to choose, at run time, between different objects that perform the same function. Users can thus select a favourite word processor to perform editing functions based on their skills and preferences, thus maximizing their productivity within an albeit standardized environment. Other common facilities, such as spreadsheets or graphics servers, may be treated in the same flexible but disciplined manner. In this way, ORBs – for all their weaknesses – address some of the fundamental points made about productivity in Chapter 1.

The CORBA standard claims to address five key problems for distributed object systems: integration, interoperation, distribution, reuse and group work. Integration is addressed by providing a standard Interface Definition Language (IDL). Hitherto, the problem of distribution had been solved using very low level RPC programming and every developer, notably those in the banks, invented their own frameworks.

ORBs are closely related to object wrappers since access to an object or package on another system takes place *via* the broker, which at once defines the protocol in IDL and locates the service. The IDL interface could be to an object-oriented

system or it could be to an entire package that is treated as if it were wrapped – provided that the IDL standard is observed by the wrapper.

CORBA Version 1, adopted in 1991, defined IDL as a protocol for inter-object communication. By 1996 Version 2 defined a way for ORBs from different manufacturers to inter-operate. The advantage of this in terms of vendor independence is obvious. IIOP is the CORBA Internet Inter-ORB Protocol. It is a lightweight transport protocol that allows heterogeneous ORB products to communicate *via* TCP/IP; that is to say it consists of a set of message formats. The OMG's General Inter-ORB Protocol (GIOP) defines a way to map IDL interfaces to messages. IIOP converts GIOP messages into TCP/IP, which means that they can be sent over networks or the Internet.

Java has its own, built-in, object request broker: RMI. Java RMI (Remote Method Invocation) is not compliant with CORBA. It allows Java applications to use the services of other Java applications as if they were local, regardless of location. To use a server written in another language the developer must create a proxy in Java on the same server and then use the JNI (Java Native Interface) to connect them. With RMI, Java acts as its own IDL. However, pure Java now also includes IIOP and a separate CORBA-compliant Java IDL.

Microsoft offers a partial alternative to CORBA in the guise of COM and DCOM (Distributed Common Object Model). These developed from OLE (Object Linking and Embedding) which allowed applications to be launched automatically when an item that they had created was accessed in a foreign document. COM assumes the immediate availability of local services but DCOM allows COM to communicate between machines using remote procedure calls. This is a far less elegant and general solution than CORBA but is much easier to implement and therefore popular with developers.

CORBA 2 also added language mappings for Ada, COBOL, C++, Java and Smalltalk and extra features for initialization, transactions, security and the Dynamic Skeleton Interface (DSI), which allows invoked operations to be selected at run time rather than be hard coded into stubs and skeletons (see below). It allowed objects to be passed by value as well as by reference and, significantly, a COM to CORBA interface.

By 2000, CORBA 3 had added support for firewalls and URLs and some features normally associated with MOM: asynchronous messaging and queuing and real-time features. CORBA 2 had relied entirely on a synchronous model based on RPCs. Version 3.0 also included the CORBA component model (CCM): a model based on Enterprise Java Beans (EJB) but language independent. COM and OLE interoperation is supported there is a mapping from this CORBA component model to Active X controls and Java beans.

ORBs allow organizations to implement service-based architectures very easily. A good example is the implementation of a volatilities publishing system at a leading wholesale bank. Here traders provide the prices of derivatives for salesmen based on proprietary algorithms and upon the volatility of the underlying instrument prices. Unfortunately, most of the requests for prices from the sales desk do not

lead to real, profitable trades. This meant that much valuable time was being wasted. The bank therefore created a system whereby the traders prepared the volatilities and algorithms using their Excel spreadsheets and a library of C++ functions callable from the spreadsheets. The results were then published to a server that could be accessed (*via* OLE) by other users of Excel or from Java enabled Web browsers across an object request broker. This bank offers many ORB-based services such as this to its internal clients. Another example is its calendar service that enables any user to find out when the various markets around the world are on holiday or closed. This service based architecture has eliminated much duplication of effort on the part of developers building such functionality into their systems.

The rôle of middleware in software architecture will become increasingly important in the coming period; object request brokers, transaction monitors and message delivery mechanisms all have a crucial rôle to play in the delivery of robust flexible systems. Unfortunately, we operate in a culture where middleware is still often jocularly defined as 'the software that no-one wants to pay for'.

One of the first commercial ORB products to emerge was Hyperdesk's ill-fated DOMS. In fact, the creators of this product were influential in setting up the OMG in the first place. They started building a replacement for Data General's ageing CEO office automation products within that company. HP's NewWave technology was unsuccessfully tried as a basis for this work. In the late 1980s the group was spun off from DG as a separate company and Chris Stone left to form the OMG. Companies such as Sun, HP, DEC and NCR all had ideas on how to solve the distributed office computing problem with objects at this time.

There were two basic approaches, a static approach originating from Sun and HP and a dynamic, but less efficient, one coming from Hyperdesk and DEC. The dynamic model requires a single request building API and a single message unpacking mechanism. In the static model there was one code stub per operation and each request was made by a different subroutine. Each object bound in skeletons specific to each operation. Each skeleton delivered parameters as if the requester were a subroutine. CORBA represents a synthesis of the static and dynamic models of object request brokers and every CORBA compliant implementation must support both approaches.

A typical, popular CORBA compliant object request broker is Orbix from Iona Technologies in Eire. Other products among the many available are ExperSoft's ORB Plus, IBM's Component Broker, Inprise's Visibroker, Peer Logic's DAIS and BEA's ObjectBroker.

Several ORBs offer interest registration facilities so that broadcast messages can be supported without the need for an explicit blackboard object. The target object is sent a message registering the interest of some other object. When an event occurs in the target object the interested object is sent a message.

Some products use a directory services table structured similarly to that of the X500 E-mail standard. References to distributed objects are obtained from the directory as surrogates that are copied to the task space of the requester. Messages

are sent to the surrogate and it relays them to the real object transparently. In fact, the distributed object manager itself is responsible for this. Figure 4.5 illustrates this remote method invocation model.

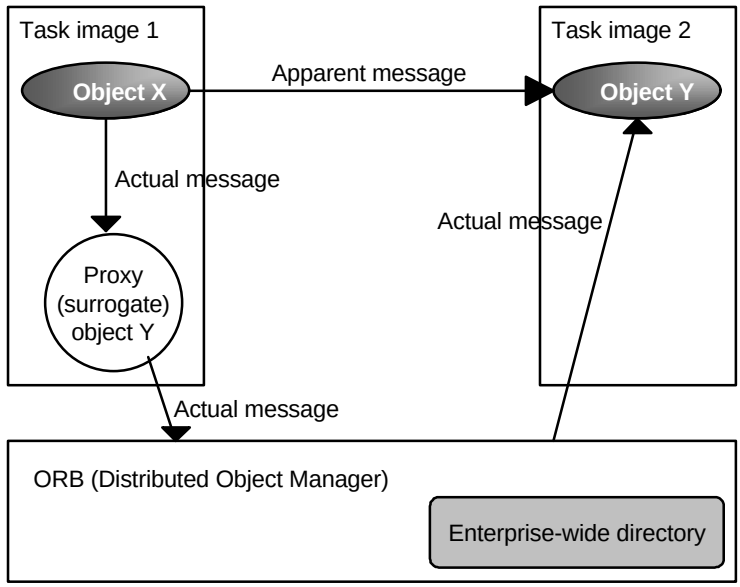


Figure 4.5 Remote method invocation

IDL is used to define the attributes and operations of interfaces (their signatures). These are compiled into stubs, skeletons and definition files. A stub is a client-side surrogate or proxy object that receives service requests locally (*via* a generated local function call), marshals the parameters and forwards the message to the real target. A skeleton then receives the invocation, unmarshals the arguments and calls the target methods directly. Skeletons must be filled out with application code.

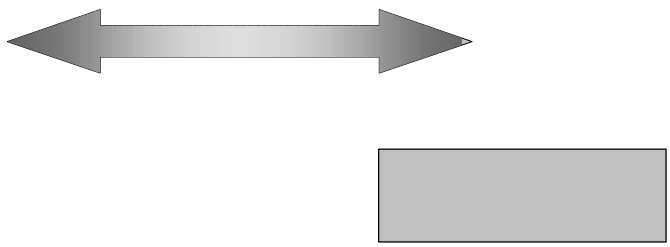
The term 'marshalling' remained something of a mystery to me until I realized that American speakers call railway goods yards 'marshalling yards'. Thus the image is that of a locomotive sorting and shunting trucks into order, ready for transmission to a new station. By analogy, when the ORB has found a suitable server, it marshals the parameters into a format suitable for transmission. Of course exceptions and error messages may be returned to the stub for relay to the client. Figure 4.6 summarizes the relationships between the various CORBA components discussed so far.

There remain problems that are not directly to do with message passing. These include concurrency, recovery from node failures, optimal object location for efficient access, how objects should be physically distributed when there are

conflicting demands from different users and whether and how large objects can be decomposed into their components and distributed. Objects can be divided into active and passive types. Active objects provide services to others, can be copied as surrogates and need concurrency control. A printer might be an active object. Passive objects may be physically distributed when their services are required. A trivial example of a passive object is a number. There are three strategies for determining the location of active objects in this framework:

1. Set the location explicitly.
2. Set the location, statically or dynamically, on the basis of the resources it needs and those available.
3. Allow the requester to set the location dynamically.

This separation of active from passive objects is useful provided that the split can be made or the objects redefined to support it, the processing requirements of active objects can be defined and mapped onto physical hardware and that the passive object mapping can be decided upon.



Since the acquisition of ObjectBroker from DEC by BEA, another middleware trend has been the integration of ORBs with transaction processing monitors. There are several TP monitors for workstations including Ellipse – Cooperative Solutions Inc., Encina – Transarc Corp., MTS – Micro Focus, TOP END – NCR, Tuxedo – part of BEA's Iceberg, and even IBM's CICS. The latter, it is reported, had its internal design heavily influenced by object-oriented ideas taken from Smalltalk and became notorious by winning a Queen's Award for the development team's use of formal specification using Z. Hitachi's TPBroker also combines a high reliability TP monitor with Visibroker, which in any case has its own transaction service. It has been used for such high performance applications as on-line bond trading. I expect that, in the future, the market will close in on a generic class of middleware offering the facilities of both ORBs and MOM with a TP monitor and real-time option. Specific industries will define semantic extensions, probably based on XML, and these may be standardized too.

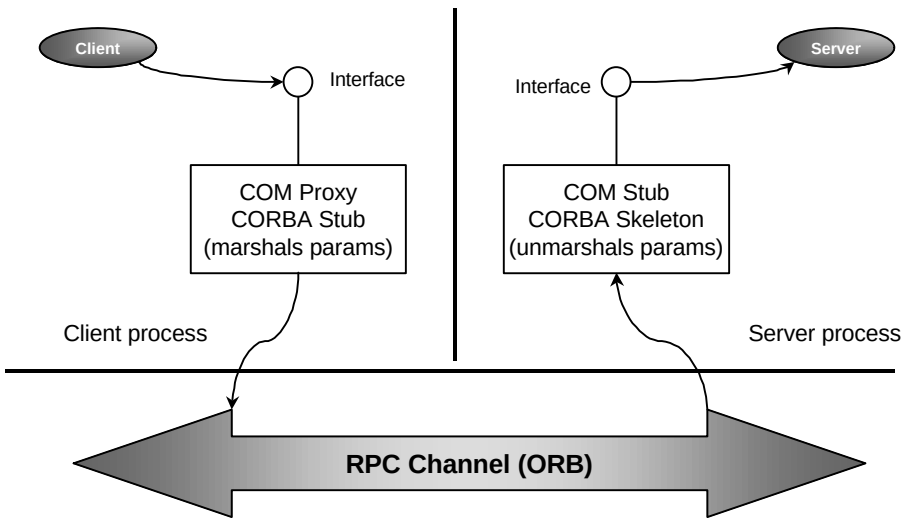


Figure 4.7 Marshalling across process boundaries in COM and CORBA

ORBs are fundamental to building service based architectures based on black box components, because of the implied lack of a single address space.

COM

The main competing standard approach to CORBA comes from Microsoft in the form of COM+, including as it does the proprietary COM, DCOM, MTS and Active X technologies. COM defines a binary standard for interfaces and so should be language independent. CORBA specifies no particular layout in memory and achieves language independence by defining multiple language bindings. However, the COM standard insists that interfaces are built in precisely the same way as C++

virtual function tables, which compromises this aim considerably. The equivalent of the ORB is called the RPC Channel. Stubs are called proxies and CORBA skeletons correspond to COM stubs. In other ways the approaches are very similar, as illustrated in Figure 4.7. For example, COM supports message queuing and event notification *via* MSMQ. COM also defines its own IDL – not to be confused with the quite different CORBA IDL.

COM+ is bundled with Microsoft's Transaction Server (MTS), which compromises it as a true cross-platform standard; partly because this and other extra services are not always supported by COM/CORBA bridges. To remedy this situation Microsoft designed the Simple Object Access Protocol (SOAP) to support component interaction across the web. This allows developers to use any 3-tier component architectures on any platform and integrate them.

Distributed computing brings with it problems additional to those of a centralized approach. For example, every user in a different department may work with a different definition of business concepts such as *customers*. A customer could be a company or a person within a company. Is a child a customer when a parent pays but the child chooses? At what point does a prospect become converted to a customer? These questions merely hint at the difficulty. With a centralized system the differences do not matter so much because users can take copies of the data they need and add their own view. The distributed totality is never assembled to reveal any discrepancies. A distributed system therefore means that it becomes necessary for all to agree on definitions; which can be very hard to do. One way that distributed applications can overcome the problems involved is to use XML (eXtensible Markup Language) on top of what ever middleware is in use.

4.2.1 The rôle of XML

XML is an extensible, generalized mark-up language that can be used to describe data passing between different systems or applications. MOM does not help with locational transparency; you still need to know where the target application lives. Therefore, a router of some sort is needed to go beyond simple point-to-point connectivity. XML is derived from the older SGML (Standard Generalized Markup Language) and is similar to HTML except that data are made self-describing by using a hierarchy of tags, making web pages closer to programs. XML permits message senders to separate content from presentation; so that rather than just saying that a piece of text that we wish to stress in set in bold type, we tag it as <emphasis> and use an XML style sheet to declare that emphasised text is bold – or whatever. For example, in the following fragment we have tags defining the structure of a message (from, to, body), its formatting (italic) and part of its content (customer).

```
<?xml version="1.0"?>
<!DOCTYPE "sales memo">
<from>Suzie Eizus</from>
<to>Eric Cire</to>
```

```
<body>We have just received an urgent order from <customer>Imperfect
Palindromes Inc.</customer> please action today.
...
</body>
```

XML parsers can enforce which tags must be present in a document (its *validity*) by reference to its document type descriptor (DTD). The fact that fields can be optional means that the recipient of an XML message with missing fields can still use the data contained. Thus, if a legacy database holds one definition of a customer, whilst a new system has another, they can still communicate. As a result, one of the chief uses of XML is information exchange between systems with different storage formats for similar information, possible across platforms. It can be used on to of middleware to create corporate standards for data interchange and to wrap legacy applications. This kind of application has become known as Enterprise Application Integration (EAI). XML also brings the potential of Electronic Data Interchange (EDI) within the budget of far smaller suppliers than was hitherto possible. For such applications, some tags are regarded as metadata.

There as several standards for metadata such as Netscape's Meta Content Framework for describing the content of website. In this context one might imagine tags such as: <Geology>, <Shakespeare> or even <useful for wannabe bomb makers>. XML is very flexible and can be used to define new, specialized markup language for various disciplines. For example it has been used to define equation editors as well as standard languages for business-to-business e-commerce. It can support multiple languages because of it use of the 16-bit Unicode character set – ASCII had only 7 bits (or 128 characters) originally[2]. XML styles sheets are usually written in another extensible language: XSL. There is also an XML user interface definition language (XUL) which uses tree structures to define UI widgets and allows them to be deployed to browsers on the fly.

The self-describing, hierarchical tag structure enables much richer data access possibilities than is normally available in databases, such as context-sensitive queries and navigation. The POET Content Management System provides an example of this kind of application of XML. A related type of application, that I have experience of, is the management of multiple foreign language translations of product labelling or documentation. Here XML allows documents to be treated as composites of smaller sections – right down to standard phrases. Combined with automatic version control, this approach cans reduce document creation and maintenance workloads by orders of magnitude. Xerox's Astoria is another product designed specifically for this kind of application. In addition to this kind of initiative there is a group of XML servers, such as ODI's eXcelon and Software AG's Tamino. These products extend OODBs to include an object model based on XML. There are plenty of XML parsers and editing tools on the market; and several are free.

---

[2] Of course, an n-bit code permits 2^n characters to be represented.

XML is actually rather too flexible. One needs to ensure that messages can map to the constructs of the languages in use while retaining maximal simplicity. One approach is to create an internal standard subset of XML, with defined tags, that all messages from legacy systems can be converted into. Chase Manhattan Bank did this (O'Sullivan, 1999). They needed to connect applications on multiple platforms that used a variety of languages, from COBOL and RPG to C++ and Java. Each system defined a different fixed-length message format. Previously, maintaining multiple point-to-point FTP transfers had been very costly and error-prone. They needed guaranteed delivery and so used IBM's MQ Series MOM product, defining its message formats with XML. This enabled them to define standard message formats for things like trades, with many optional fields – supersetting those of existing formats. Although CORBA was also widely used at Chase, it could no more provide the message transmission format than could MQ Series or notations like ASN.1. None of these were readable by humans either. Chase defined several restrictions that would be built into their parser. These included several 'type safety' rules, such as not allowing message elements to be declared as ANY or EMPTY and insisting that leaf elements declare their type. This is a typical application of XML and similar projects in other banks will lead to inter-industry standard languages of this type. the Open Financial Exchange (www.OFX.net) is a set of standard financial element types representing things like credit card numbers and transaction types. The Open Trading Protocol (www.OTP.org) is another standard for more general electronic trading.

One rather whacky application of XML allows a virtual newsreader's facial image to smile, frown or pout appropriately while speech is synthesised from news text. The expressions are just coded as tags embedded in the text. Her tone of voice is also influenced in this way. This extends the idea of *avatars* – fairly common in chat rooms now but originally predicted in a wonderful science fiction novel (Stephenson, 1992).

Microsoft's Biztalk is a proprietary XML-based server framework that many companies have used as the basis for applications like supply chain integration. Biztalk includes schemata for such business processes as product catalogues, offers and purchasing. It routes messages and can include workflow rules. XMI (XML Metadata Interchange) is an OMG standard that attempts to unify UML, MOF and XML. It uses XML DTDs to describe MOF components and UML models. MOF-compliant model will be interchangeable between XMI-compliant repositories.

XML looks set to be the universal language for self-describing messaging between disparate systems and entire businesses. As long as XML doesn't just mean that even more enterprise programs are able to misunderstand one another successfully!

4.3 Enterprise Application Integration

The 1990s saw many organizations undergoing major changes in technology, principally moving away from systems based on centralized mainframes – usually from a single supplier – to distributed systems. Workstations became desirable platforms for both development and applications due to their increased and more accessible power and the ease of building usable graphical interfaces on them. This was associated with a greater emphasis on rapid response to changing business needs and the use of rapid development. The existence of immovable legacy systems meant that interoperability was a key issue during the migration. Developments in hardware and relational database software also led to increased demand for, and reliance on, distributed architectures. Typically these organizations operated with a goulash of mainframe systems, departmental minis and standalone or networked PCs running word processors, spreadsheet applications and, at the end of that period, browsers. The mainframes ran both relational and non-relational databases and record-oriented applications written in COBOL or Assembler.

As new applications began to appear the first need was for them to access information on the mainframes. This could be accomplished by nightly downloads but the more advanced solution was to build a bridge using open gateway software that could convert between systems such as IMS on the mainframe and Oracle or Sybase on UNIX servers. Often this software was supplied by the relational database vendors themselves, a typical early product being Sybase's NetGateway. In this way, data could be both retrieved and updated rather than merely copied; thus removing the necessity to rebuild terminal data entry screens for new applications. This approach was found to offer considerable locational transparency to users and made database interaction far easier.

One of the complex migration issues that was faced by most organizations was that of network architecture, resilience and management. One organization I worked with viewed its networks as offering a hierarchy of service layers. The highest level enabled users to connect their workstations to various local servers. The next level provided workgroup server support. Both these layers relied chiefly on LAN technology and the workgroup level used matrix switching. The third layer linked the previous two to departmental and database servers using Ethernet switches. Finally, there was a massively complex connectivity layer that supported gateways to the mainframe-based legacy systems, access to WANs and message routing. Security is an important issue in the financial services sector and this company took it very seriously indeed. They adopted the Open Software Foundation's DCE security services standard, Kerberos, within their network management system. Routers were employed to connect different LAN systems such as Ethernet and Token Ring and to provide firewalls between network segments. Within two years their distributed computing architecture enabled this company to construct several complex, business critical and flexible applications.

Typical systems moved trading data from a mainframe network database and combined them with prices from a market data feed. The data were placed in server-based relational systems, validated interactively and then returned to the mainframe and passed simultaneously to local workstations. Other systems provide seamless 'straight-through' links between previously incompatible front and back office systems.

Today EAI is the name of the integration of disparate systems, packages and components into key business processes; i.e. making the different chunks of software within an enterprise work together – ideally as well as if they had been designed as components in the first place. EAI projects therefore need many of the tools we have been discussing: middleware, metadata, message formats, etc.

An organisation and its software evolve over a number of years and many point-to-point connectors have been defined between software packages, for specific purposes. The overall system (and therefore the business) is inflexible because the connections cannot be rearranged without huge effort; the connector definitions are specific to the modules they connect. Some couplings that would be advantageous have never been made. A similar situation arises when legacy systems have evolved in separate departments, or in different subsidiary companies; the move towards e-commerce now demands that they be integrated more closely than before. The solution is to define a set of common connectors – a language that all the components can be adapted to talk. It is then easier to rearrange them. This is not just a matter of choosing a technology such as MOM, CORBA or XML in addition one must decide upon the protocols and common business model needed. Not is it solely about data integration and warehousing across disparate systems; the functions of the systems must be integrated too.

Competitive pressures, new ways of working and the move to e-commerce imply that there is a need for real-time data and process integrity; overnight downloads and replication are no longer enough. Not only must processes cross organizational boundaries but work may flow across several different companies. All this must be implemented ever more rapidly. Not only data formats must be transformed but business concepts too: converting a 2-digit date to a 4-digit one is a fairly tractable task, but mapping customers that have multiple locations to a system that thinks there can only be one is far harder. Package software is usually too inflexible and incomplete to provide the solution. EAI requires attention to the development of common business standards and a unified technology architecture. It also needs clear, separate interfaces to modules (APIs). Package vendors will claim to supply all this, but – of course – only if you buy most of what you need from them! That route may well lead to next year's legacy integration problem.

In practice, one may have to decide on a core interface technology: standardizing on one of Microsoft COM interfaces, EJBs or CORBA. But even then full consideration has to be given to more than just the technology architecture. Figure 4.8 shows the provenance of various technologies and concepts in relation to architecture.

EAI software products should support event notification, workflow management, rule-based message routing and data transformation. The latter implies that there be tools to define interfaces and data format transforms and a repository to store all this. Vendors that operate in the EAI space include sellers of packages, middleware and wrapping tools, workflow automation tools, database gateways and application servers. Application servers range from integrated e-commerce solutions such as IBM's Websphere and BEA's Weblogic to database caches. There are also vendors of application development tools such as Compuware's UNIFACE and Sun's Forté Fusion. Fusion builds on Forté Conductor – a cross-platform, component-based workflow management and integration system – and the OO 4GLs: TOOL and SynerJ. It uses XML for messaging and XSL for presentation and process rules. Systems are linked through application proxies that talk to native language APIs – such as C, TOOL or XML – *via* connectors.

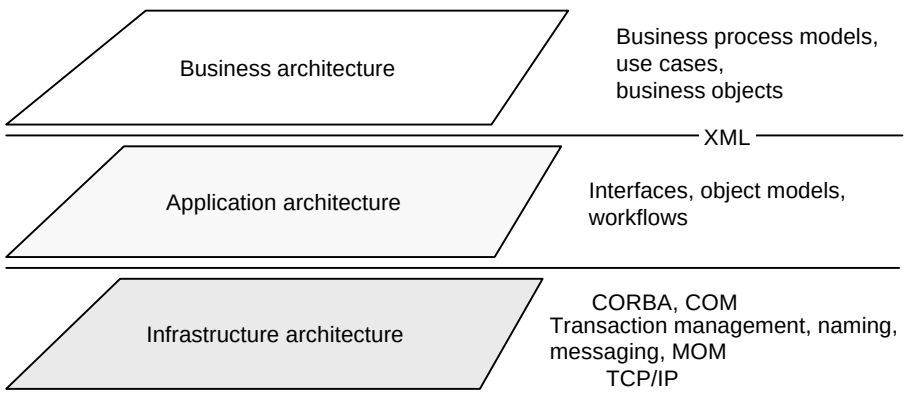


Figure 4.8 Architectural layers

Introducing the glue that joins up the legacy may of course affect performance and make testing and locating faults harder, but that is the price of increased flexibility in the face of business change

EAI products do not provide an off-the-shelf solution; interfaces, processes and new components will always need to be developed and/or tweaked, and this can be a significant development task. Doing it properly requires honed skills in software architecture and component based development. In many ways EAI is just a different take on the same set of concerns as CBD, using the same techniques. In CBD, we think of constructing pieces that be glued together easily; in EAI, we think of adapting existing pieces so that they work together; in practice, most developments are a mixture of the two. We will look at techniques for CBD in Chapter 7 in some detail.

The key technical features in an EAI environment are distribution, connectivity and flexibility. These factors have led away from the waterfall-structured-

mainframe mind-set and people have realized that not only are advanced, object-oriented programming environments essential to exploit evolution and describe a parallel, distributed, message-based world but also they would have to adopt object-oriented analysis and design approaches. Without sound object-oriented analysis and a project management culture that accepts its implications, the distributed solution will soon become even more cumbersome to manage than the mainframes it replaced. The key task for companies engaged in EAI is the definition of such a sound approach.

4.4 Migration strategies

Many people and organizations are convinced of the wisdom of shifting their systems development activities towards an object-oriented, component based development style. This may be because they have seen other companies succeeding in this way or even for that worst of reasons: because OT and CBD are fashionable. Even in the latter, misguided, case these companies may gain from the experience because, even should the project in hand fail, they may gain a better understanding of existing systems and development practices through the construction of an object model. They have several reasons for replacing or extending older systems. For example, a package vendor may see the move to object technology as closely tied to the move to a new platform and, in turn, see this as a way of achieving greater market share. They may wish to compete more effectively by adding value to the existing product with graphical user interfaces, management information system (MIS) features or delivery on distributed platforms. User organizations may wish to take advantage of new standards, friendlier interfaces or opportunities for e-commerce along with the benefits of the move to OT itself that were discussed in Chapter 2. In the latter case it is essential that their e-commerce systems – whether for trading with other businesses, selling to customers or integrating the supply chain – are almost infinitely scaleable from the outset, since the number of users is quite unpredictable. Also, there must be total flexibility in the face of requirements that are bound to change rapidly. This implies the flexibility of object technology as well as the need for a sound requirements engineering process: one compatible with both objects and business process change. New e-commerce companies (or subsidiaries) will not face huge legacy integration problems but will have to build up every capability from scratch. One the other hand they do not face the predictable problems of resistance to changing business practices and culture faced by established companies.

Both vendors and users will be looking to slash maintenance costs, which can account for a huge proportion of the IS budget, and reduce time to market. We have already seen that OT can contribute to both goals if implemented successfully. However, while overnight migration is highly desirable it is seldom possible. Furthermore, gradual migration may take too long for its benefits to be worthwhile.

Often the solution is to reuse existing conventional components or entire systems and packages; through EAI. Also, some accounting and ERP package vendors have attempted to 'componentize' their offerings so that one can take modules from competing products and make them interoperate. Will discuss this development separately in Chapter 7 when we deal with component based development. There are several available options: inter-operation, reuse, extension, and gradual or sudden migration. These options are closely related but we will deal with inter-operation first.

In this section, we examine various proposed and actual strategies which meet the requirements of organizations facing migration and inter-operation problems, emphasizing the concerns of a developer who wants to develop an object-oriented application that needs to use the services provided by applications that incorporate other programming styles such as expert systems, 4GLs, procedural libraries (e.g. the NAG FORTRAN library), parallel processing systems, relational databases or even fuzzy controllers. We must ask what the fundamental issues are in using, designing and building inter-operation tools. How do you deal with those critical COBOL or Assembler applications? What is the rôle of object-oriented analysis and component-based design techniques within this kind of migration? Is there a strategy that enables you to metamorphose an existing procedural application into an object-oriented application without disrupting services to the existing users? What is the rôle of EAI tools?

This section will also cover such problems as how to wrap an old application which exists in a large number of different versions. We will also explore the ways in which object-oriented analysis in particular and object technology as a whole can be used as a migration technique.

4.4.1 Interoperation of object-oriented systems with conventional IT

There are a number of scenarios in which an object-oriented application should inter-operate with existing non-object-oriented systems. These include:

- the evolutionary migration of an existing system to a future object-oriented implementation where parts of the old system will remain temporarily in use;
- the evolution and integration of enterprise systems which already exist and are important and too large or complex to rewrite at a stroke and where part or all of the old system may continue to exist indefinitely (EAI);
- the need to build on existing package solutions (a second context for EAI);
- the reuse of highly specialized or optimized routines, embedded expert systems and hardware-specific software;
- exploiting the best of existing relational databases for one part of an application in concert with the use of an object-oriented programming language and/or object-oriented databases for another;
- the construction of graphical or browser front-ends to existing systems;

- co-operative processing and blackboard architectures may involve agents which already exist working with newly defined objects;
- the need to co-operate with existing systems across wide and local area networks and the Web.

The main issue is how to tackle the migration of a vast system that is almost invariably very costly and tricky to maintain. The first strategy recommended here is to build what I have referred to several times as an OBJECT WRAPPER. Object wrappers can be used to migrate to object-oriented programming and still protect investments in conventional code. The wrapper concept has become part of the folk-lore of object-orientation but, as far as I know, the term was first coined by Wally Dietrich of IBM (Dietrich 1989) though it is also often attributed to Brad Cox and Tom Love, the developers of Objective-C, but in a slightly different context. There are also claims that the usage was in vogue within IBM as early as 1987.

The existence of large investments in COTS (commercial, off-the-shelf) packages and programs written in conventional languages such as Assembler, COBOL, PL/1, FORTRAN and even APL has to be recognized. It must also be

system's functions and thereby access its data too[3]. Effectively, the wrapper is a large object whose methods are the menu options of the old system. The only difference between this new object and the old system is that it will respond to messages from other objects. So far, this gives little in the way of benefits. However, when we either discover a bug, receive a change request or wish to add a new business function the benefits begin; for we do not meddle with the old system at all but create a new set of objects to deliver the new features. As far as the existing users are concerned, they may see no difference when using the old functions; although their calls are being diverted *via* the wrapper. Wrappers may be small or large, but in the context of inter-operation they tend to be of quite coarse granularity. For command driven systems, the wrapper may be a set of operating system batch files or scripts. It could also be a set of CORBA or COM+ IDL interfaces. If the old system used a form or screen-based interface, the wrapper may consist of code that reads and writes data to the screen. This can be done using a virtual terminal. This is fairly easy to accomplish on machines such as the VAX though it is not always possible with systems such as OS/400 where some specialist software, an object request broker or Java RMI may be required. All new functions or replacements should be dealt with by creating new objects with their own encapsulated data structures and methods. If the services of the old system are needed, these are requested by message passing and the output is decoded by the wrapper and passed to the requester.

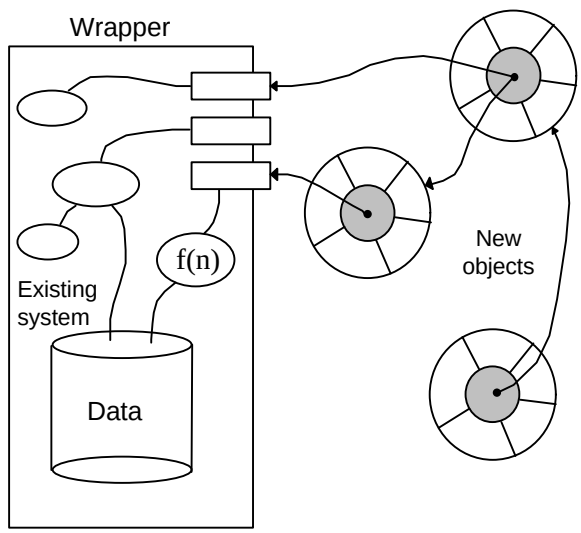


Figure 4.9 Object wrappers

[3]The term *Gradygram* was coined to stand for the icons with operations indicated by small boxes on the boundary of a rectangle representing the object used by Grady Booch since his work on design for Ada in the 1980s. UML uses them for its module diagrams to this day.

So much for the propaganda! Implementing wrappers is not as easy as it sounds in several respects. Much of the literature on wrappers is aimed at deriving the necessity of object request brokers. When these are not available, for whatever reason, developers have to face up to the implementation details directly. One such issue concerns granularity. Most of the theoretical arguments and a good deal of the practical experience of object-oriented programmers indicate that small objects are usually more likely to match particular requirements than large ones. Recall the guidelines for object design from Chapter 2: interfaces should be small and simple; no more than about 17 operations per object; and so on. However, with the legacy system, or with large grain components, we are faced with a *fait accompli*; the system is as it is. There are irreducibly large grain 'objects' often present in the design of legacy systems. Object request brokers and component models are specifically aimed at dealing with this kind of coarse grain reuse. The question is whether, without such a broker, we can still gain from the use of a hand-made wrapper. Some developers find that coarse grain components arise naturally even with new requirements and deduce that object-oriented models are not always appropriate. Brice (1993) for example found this in the context of geometrical image transformation software. The data structures were straightforward but the processing required pages of equations to describe it and data flow models were found to be the most natural thing to use. Here is a case where home made wrappers may be beneficial even with green field developments.

Component based development also emphasises coarse grain components consisting of many classes implementing several interfaces; so that any component is *a fortiori* usually a wrapper.

This approach to migration is not the only one available. Other options include the use of object request brokers (ORBs) and proceeding in a completely *ad hoc* manner. The *ad hoc* approach is often the correct one, but there are so many ways of approaching a particular problem that few sensible generalizations can be made. The *ad hoc* approach was the only one available until the appearance of ORB products and message-oriented middleware. Now it is very common to use an ORB to implement a wrapper around a legacy system. One leading financial institution, for example, built a straight-through trading system to connect its front and back office systems using objects that effectively comprised self-describing data packets. The result was observed *post facto* to be an application specific object request broker but the work was completed before any stable commercial CORBA compliant distributed object management system had come to market. Furthermore, this application went beyond the CORBA specification by using self describing data, as one would now do using XML.

One of the biggest problems with the concept of object wrappers concerns data management. Using the wrapper is easy until you need to split the storage of data across the old database and some of the new objects.

4.4.2 Data management strategies for wrappers

The object wrapper approach seems ideal at first sight but closer examination reveals some severe data management problems. Where building a wrapper makes it necessary to duplicate or share data across the new and old system components, there are at most four possible strategies:

1. Carry a duplicate live copy of the common data in both parts of the system and keep both copies up to date. The problem with this is that storage requirements could double. Worse still, there are real integrity issues to worry about. It is almost certainly not a viable strategy for either migration or reuse of any commercial scale system. We will call this the **tandem** or **handshake** strategy because it requires constant synchronization of updates and retrievals. It only works when there is little or no overlap between the data of the old and new systems, which is rare.
2. Keep all data in the old system and copy them to the new objects as required. Messages to the old system cause it to handle updates. This is known as the **borrowing** or **download** strategy because data are borrowed temporarily from the wrapper. It is similar to what is done in many existing, conventional MIS applications where data are downloaded nightly from a mainframe to workstations and updates transmitted in batch. A more sophisticated variant of this strategy is common when object-oriented programs are linked to large relational databases. Data from the database are cached in an object-oriented database so that the programmers do not have to concern themselves with the object-to-relational mapping and performance is enhanced (for reasons to be explained in Chapter 5). This variant is the **caching** strategy and application servers often employ it – making the technicalities transparent to programmers.
3. Copy the data to the new objects and allow the old system's data to go out of date. Again there may be integrity problems, and the wrapper may have to send messages as well as receive and respond to them; which greatly increases its complexity. Call this the **take-over** strategy by analogy with a company making a take-over bid for another.
4. Carve out coherent chunks of the database together with related functions. This is difficult and requires a sound method of object-oriented analysis capable of describing the old system as well as the new and/or a translation technique from original systems design documents such as DFDs. On balance, it seems the most promising approach to legacy migration. This is called the **translation** strategy because one must translate the design to an object model. It is easiest to do when the old system was originally written around critical data structures using a technique such as step-wise refinement. These structures and the programs that use them will migrate naturally to the objects of the new system. A refinement of this strategy is to reverse engineer a data model from the existing system and to identify all

file access operations in relation to this model using, for example, a CRUD matrix approach. CRUD techniques are often used to organize conventional systems around their data structures. These calls can then be replaced as new objects are constructed around the entities. This improved version of translation can be called **data-centred translation**. Whether it is feasible will depend on the difficulty of obtaining a data model and the complexity of the code in which the database calls are embedded. Reverse engineering tools may prove useful within this strategy and, as Reiss (1991) points out, the most useful tools would contain an understanding of the system semantics.

These strategies may be variously appropriate according to whether we are migrating the system to an object-oriented implementation, reusing its components, extending it or building a better or a distributed front-end. Assuming that our chief aim is to migrate the old system to a new object-oriented one rather than merely to reuse it components, which strategies are feasible? The handshake strategy is flawed for all but the smallest systems, and then there must be little overlap between new and old components. The borrowing strategy may well involve tampering with the old system and is not usually viable for the purposes of system migration unless there is a clean separation between existing functions and new requirements – though it is suitable for enabling legacy reuse. Borrowing does not permit data to move permanently outward across the wrapper boundary. This means that there will come a time when a huge step must be taken all at once to migrate the data out, unless a DBMS has been used for all data accesses. These strategies, as **migration** strategies, do grievous violence to the whole idea of building wrappers. Only the last two strategies promise to be feasible if our intention is to migrate the functions of the old system to a new one, and there are some systems where neither seems to be practical. It is also the case that the type of system, its structure and the quality and type of its documentation will affect the choice of strategy. Dietrich's original application of the wrapper concept was to a solid modelling system of considerable complexity but whose intricacy resided in its code rather than in its data management. Furthermore, his primary concern was with the reuse of the functions of a stable system rather than its reconstruction.

Strategy 4, translation, will work most often, provided that the old system can be decomposed around coherent data sets and if there are, say, some existing DFDs to transform objects from by encapsulating their data stores. If not, one is faced with building a wrapper of much greater complexity using take-over: strategy 3. The latter is a far costlier option.

Critical to all this is the adoption of an architecture-centric approach and the use of patterns and component kits based on sound interface specification techniques. We will see in Chapter 7 how recent advances in object-oriented design such as Catalysis (D'Souza and Wills, 1999; Wills, 2001) contribute significantly.

4.4.3 Practical problems with migration

Another problem arises when the old system exists and is maintained in multiple versions. For example, a commercial package for a particular industry may, over the years, have been adapted for the needs of particular clients. The cost of building a wrapper for each version is usually prohibitive. The wrapper approach will only work if there is a core system common to all the versions, and the modifications will have to be maintained separately in any case until they can be re-implemented. This was the situation on a project that I was involved with where there were around 70 versions of the product customized for particular sites scattered across the globe, with local, dedicated maintenance teams in many cases. Also, the decomposition of the existing system into coherent chunks was exceedingly hard because of the long modification history. The strategy adopted was to model the system using object-oriented analysis and first wrap the core system in such a way that new functionality (an MIS component) could be added using object-oriented methods, leaving the core system much untouched at this stage. The AS400 hardware on which the system had to run, pending a move to UNIX at some future date, did not then support an object-oriented language of any form. Thus, in the short term, the new object-oriented components had to be implemented in a conventional language. To ensure that the new system can be fully object-oriented in the future we had to find a way to minimize the cost of so doing. This led to the use of an object-oriented analysis approach and the conversion of its products to conventional code. It also led to the use of a (now defunct) middleware product, NeWI, that was able to let the developers treat the system as object-oriented while still writing code in C. We also produced an object-oriented description of the existing system to clarify understanding and help carve out separate re-implementable chunks following a translation strategy. It emerged that treating some functions as objects instead of methods was useful. The bulk of the early effort went into designing additional features and their interface, *via* a wrapper, to the core of the existing system; translation tasks being deferred to the near future. Thus, it proved wise to proceed in steps:

1. build a wrapper to communicate with new object-oriented components using (most probably) the borrowing strategy;
2. perform an object-oriented analysis on the old system;
3. use translation or data centred translation to migrate;
4. utilize an ORB to implement.

Grass (1991) argues that wrappers work well for mature systems that are essentially frozen, in the context of a requirement to reduce the maintenance burden which she characterizes as ‘extremely aggravating’ with the panache of understatement. Her main point is that ill-structured legacy systems are costly to understand and wrap. However, even this is worthwhile if the potential maintenance savings are large enough. Like Dietrich’s, Grass’ principal

application (a parser for regular grammars) was complex functionally but not primarily a data intensive application.

One may conclude that the best approach to migration of legacy systems with significant data management complexity is to build wrappers that support object-oriented front-ends and to build the required new functions within the front-ends. The tandem strategy can be used only when there is little overlap and separate databases will have to be maintained. The exceptions to this are when the existing system already has a coherent data centred structure that facilitates translation or when the benefits of the migration are large enough to justify the cost of building a very complex wrapper along the lines of the take-over strategy. If there is an existing DBMS this can be wrapped as a whole and maintained for a long time as the wrapped functions are gradually migrated. Then, at some point, one can move all the data at once to an object-oriented database if desired and eliminate the database wrapper. This is a special variant of the translation strategy where the database is one huge 'coherent chunk'. It is probably the ideal option for many organizations already obtaining satisfactory performance from their relational databases.

Having decided to build a wrapper and a new front-end, one needs tools for building them. Some vendors, such as MicroFocus, offer products for wrapping COBOL applications and there are several ORB and middleware tools that address the issue.

A key problem faced by many IT organizations is one I have often heard called the *goulash* problem. It exists where there is a mixture of essentially incompatible hardware and software that somehow has been made to work together over the years. Conceptually it is easy to see that this goulash can be modelled as a system of large components communicating by passing messages with parameters. The wrapper approach is appropriate when just one of these systems is to migrate to an object-oriented platform. Rather than build a wrapper for each old system, which would be expensive to say the least, it is better to wrap the communication system in some way. As we have seen, the most common approach to this problem is to use object request brokers and component based development. Several Enterprise Application Integration tools are based on middleware together with object-oriented databases and XML technology.

In the product migration project referred to above, it turned out – for the reasons given – that there were no suitable software tools at the outset. Therefore our main tool was our object-oriented analysis approach itself.

We need to deal not only with the evolution of existing systems which are important and too large or complex to rewrite but also with the evolutionary migration of an existing system to a future object-oriented implementation. This implies the need for techniques that will let us reuse components of existing functionally decomposed systems or even entire packages within our new or evolving object-oriented systems. To this issue we now turn.

4.4.4 Reusing existing software components and packages

So far, we have considered the wrapper technique from the point of view of migration. Now we must consider also the problem of reusing existing components when there is no explicit need or intention to re-implement them in an object-oriented style. Dietrich's (1989) work showed in principle how the reuse of highly optimized algorithms or specialized functions could be accomplished using the object wrapper strategy. This can be done by defining application level classes whose methods call subroutines in the old system. The legacy systems can be wrapped in groups or as individual packages, with the latter option offering greater potential for reuse. The more recent efforts of ERP vendors to 'componentize' their offerings reduces the amount of work needed to create wrappers greatly but requires that users upgrade to the latest versions.

There is also a need to build on existing 'package' solutions. Once again a wrapper that calls package subroutines or simulates dialogue at a terminal can be built. The alternative is to modify the packages to export data for manipulation by the new system, but this fails to reuse existing functional components of the old one. Also some package vendors may not be prepared to support or even countenance such changes.

Some problems which must be solved in building such a wrapper are identified by Dietrich as follows.

- The designer is not free to choose the best representations for the problem in terms of objects since this is already largely decided within the old system. Here again there is a possibility that the wrappers will represent very coarse grain objects with limited opportunities for reuse.
- The designer must either expose the old system's functions and interface to the user or protect him from possible changes to the old system. It is very difficult to do both successfully. Generally, one should only allow read accesses to the old system, which tends to preclude the take-over data management strategy.
- Where the old system continues to maintain data, the wrapper must preserve the state of these data when it calls internal routines. This militates against the translation strategy.
- Garbage collection and memory management and compactification (where applicable) must be synchronized between the wrapper and the old system.
- Cross system invariants, which relate the old and new data sets, must be maintained.
- Building a wrapper often requires very detailed understanding of the old system. This even more true when migrating but still a significant problem when reusing.

Because access to the internals of package software is seldom available at the required level of detail, the wrapper approach described above will not usually

work. A better approach is to regard the package as a fixed object offering definite services, possibly in a distributed environment.

Whereas data centred translation is the best approach to migration and replacement of existing systems – while borrowing strategies fail to work – where reuse is the main concern, borrowing is a perfectly viable approach. If the existing system or package largely works, for the functions it provides, and can be maintained at an acceptable level of cost (however large that may be) then when new functions are required it may be possible to build them quite separately using an object-oriented approach and communicate with the old system through a wrapper. This wrapper is used to call the services of the old system and give access to its database. New functions are defined as the methods of objects that encapsulate the data they need, insofar as they are new data. When data stored by the old system are required, a message must be sent to the wrapper and the appropriate retrieval routines called; borrowing the needed data. Updates to the existing database are treated similarly – by lending as it were.

It may well turn out that, in the fullness of time, the new object-oriented system will gradually acquire features that replace and duplicate parts of the old system. Data centred translation then becomes necessary instead of borrowing for the affected parts of the system. Therefore the step-by-step strategy recommended in the last section is indicated for many commercial systems projects.

We may summarize the conclusions of this section so far in Figure 4.10. In this table a 'Y' indicates that a wrapper data strategy may be worth considering for a particular class of problem but not that it is guaranteed to work. An 'N' indicates that it probably will not be suitable. A '?' means: 'It all depends.' The four strategies defined in Section 4.3.2 are compared with four possible reasons for building wrappers: migrating a complex legacy system to a new object-oriented implementation; reusing its components without changing the core system; extending its functionality without changing the core and building a possibly distributed front-end to provide additional functions. Note that the last three purposes are very similar and the last two have identical Y/N patterns.

	Purpose			
Strategy	*Migration*	*Reuse*	*Extension*	*Front-end*
Handshake	N	N	N	N
Borrowing	N	Y	Y	Y
Take-over	?	N	N	N
Translation	?	N	Y	Y
Data-centred translation	Y	N	Y	Y

Figure 4.10 Suitability of migration strategies for different purposes

4.4.5 Using object-oriented analysis as a springboard

The burgeoning of interest in object technology in the 1990s led to the widespread use of object-oriented programming languages for software development. With this growth came a realization that structured methods not only failed to support such developments but actually impeded them in many cases. Typical applications where this applies are GUI development, distributed computing, e-commerce and use of object-oriented and object-relational databases. The need for a disciplined approach to such developments led to the development of methods for object-oriented analysis. Another area where there is a need for methods is expert system development, and here object-oriented approaches were strongly indicated by the presence of inheritance structures in frame based expert systems.

In the case of inter-operation between object-oriented and conventional system components we have already seen that it is often necessary to reverse engineer an older system in order to find a description of it in terms of coherent entities. Some technique of object-oriented analysis is clearly required for this, especially if migration to OT is contemplated. Thus, I am suggesting that object-oriented analysis can be used as a fast path to the object-oriented future in the face of inter-operability problems and still evolving language technology.

THREE SOURCES OF OBJECTS

There are three distinct ways in which the concept of objects has entered computer science. The programming language community developed the object-oriented perspective from its own concerns with simulation, user interface development and so on. Database theorists in need of abstract models to help model data introduced various semantic data modelling techniques, the best known being that of Chen – the Entity Relationship (ER) model (Chen, 1976). These models dealt with abstract objects and inheritance but focused on the data aspects of entities, ignoring the procedures which entities might be able to perform. The artificial intelligence community went a step further with their concept of frames and allowed entities to have procedures attached to attributes that can be used to search for missing values and cause side effects, but frames are static data structures and offer no specific facilities for specifying the behaviour of entities declaratively. These procedures are often external though linked to the frames and not encapsulated as such. Further, frames are closer in spirit to prototypes than objects. This means that there are no definitional classes with instances that inherit all their features, merely instances that are more or less prototypical of some concept. The solution is to allow classes to be treated as prototype definitions; that is the instances may not have all of the features of the class as in object-oriented programming. In this way a three-legged, alcoholic dog can be an instance of dogs, which have four legs and drink water from choice. Languages such as SELF (Ungar and Smith, 1987) are based on prototypes rather than classes. Incidentally, this means that if object-oriented analysis techniques are to be truly language independent they must allow the modelling of prototypes by allowing overriding at the instance level. If we permit such relaxations, the object technology movement offers the richest set of means to

overcome the obstacles of the semantic data modeller, the knowledge based systems builder and the programmer alike. Let us take a brief look at each of these perspectives.

SEMANTIC DATA MODELLING

This perspective derived from practical database applications as well as from research efforts. It is at the root of the client-server metaphor and of database triggers and server-embedded business rules. Semantic data modellers use rules and inheritance structures extensively but have not been greatly concerned with the issue of encapsulation. There has been considerable cross-fertilization between this field and object technology. See for example (Gray *et al.*, 1992). The semantic data modelling perspective is concerned only with data and is largely encompassed by the other two.

OOP

The techniques of object-orientation are mainly derived from work on the programming language Smalltalk and its environment, developed at Xerox PARC during the 1970s. There are now many object-oriented languages including C++, Eiffel and Java. Their developers had various programming-theoretical concerns such as speed, correctness, type safety and denotational expressiveness. As explained in Chapter 1, in the object-oriented style an 'object' is an entity with three features: *unique identity*; *encapsulation* of attributes and methods; and *inheritance*. Encapsulation means that objects (either classes or their instances) consist of an interface of attributes and methods together with a hidden implementation of their data structures and processes. As we have seen, the key benefits of this approach are that objects are inherently units of reusable code and that systems built from them are extensible using inheritance to add new or exceptional features to a system.

AI

A frame, on the other hand, is usually described as having a set of attributes which themselves can be attached to procedures although these procedures are not usually stored with the object as in object-oriented programming. Interestingly, this is also the approach taken in most object-oriented databases. The difference between frames and objects is the lack of emphasis in the former on the encapsulation of the data and procedures together, hidden behind an interface. Secondly, in AI, the procedures may be regarded as non-procedural rulesets instead of procedural code. Another difference between object-oriented and AI approaches is that AI work has provided very rich methods for dealing with inheritance compared to object technology (OT). Nevertheless, the similarities outweigh the differences, and someone wishing to take advantage of the benefits of object-orientation (i.e. reuse and extensibility) might as well proceed using a Nexpert as a Smalltalk. The advantages are the richer inheritance features and the ability to express relationships as rule sets. The disadvantages are that the level of encapsulation (and thus reuse) is lower and, for some applications such as GUI development or CASE tool construction, existing class libraries or database managers may make the pure object-oriented approach more productive.

4.4.6 Object-oriented analysis and knowledge based prototyping

Any decent object-oriented analysis approach should be language independent. The disadvantage is that the specification cannot be tested and that the system cannot be delivered and used. The answer to both problems is prototyping. A prototype can be used to agree the specification with users and can even be implemented incrementally, pending a final rewrite in the object-oriented language of choice should performance or some other factor make this desirable. However, what we build as a model in the prototyping language needs to be *reversible* in the sense that it is possible to discover the intentions of the analysts and designers by examining the code. This should also be true of the paper analysis itself, leading to a requirement for a semantically rich analysis language and a readable, declarative, prototyping one.

The two areas of IT where such tools have become available are expert systems and advanced databases; themselves under the influence, it has to be said, of artificial intelligence. Expert systems offer declarative, semantically rich programming languages. Advanced databases allow business rules to be coded as production rules and database triggers. So far, only Illustra and the semantic databases offer anything like this. Even object-oriented databases fall short in this respect. It seems that this situation will change and the database community are certainly aware of the problem. Furthermore, several database workers now accept that business rules should be attached to the entities or objects of the data model. As Andleigh and Gretzinger (1992) put it: 'Clear documentation of the business rules *and which objects each rule is associated with* ensures that the application of the rules is perceived clearly and is well understood by the user before the coding of the rules.' (my emphasis).

We may conclude that knowledge based systems and advanced database products, or perhaps both in concert, can help build reversible prototypes. However, this is best done in conjunction with a machine independent analysis exercise using a semantically rich and therefore reversible notation.

This was confirmed within my personal experience by the ARIES expert systems project (Butler and Chamberlain 1988) where we built systems on a workstation with the conscious intention of transporting them to PCs at a later date. At every stage, including prototyping, implementation independent representations or, more succinctly, paper models were built. Porting then became very easy indeed because we ported not the code but the paper model; the paper model was reversible because it was written in the expressive language of knowledge engineers and not the irreversible products of software engineering.

As expert systems software has advanced and moved closer to object-orientation I have come to regard knowledge based systems as first rate prototyping tools in their own right. This is mainly because of their excellent and expressive inheritance features. We are quite happy to see this removed at implementation time to enhance reusability, but for analysis inheritance is indispensable.

Knowledge based systems are also inherently reversible because they express knowledge as rules and objects and not in procedural code or obscure diagrammatic conventions. The use of such an approach to specification and prototyping requires

Even experts in object technology argue about how it can be achieved and managed successfully.

The object-oriented metaphor is capable of describing the distributed nature of systems in a very natural way since message passing is central to both. Furthermore, the fact that object technology offers a way of managing complexity through encapsulation and inheritance means that it at least becomes possible to think about very complex distributed architectures.

Object-oriented tools for GUI development both make it easier and impose a style and consistency that itself contributes to usability. They enhance productivity enormously and can be used to impose standards for user interface design and promote reuse in this area. Similarly, object-oriented tools for expert system construction are beginning to be common, capitalizing on inheritance schemes to reduce the complexity of rule-bases and thus deliver friendlier, more natural dialogues between people and their applications.

It has become clear that experienced COBOL or RPG programmers *can* learn the object-oriented approach and several companies have proved this in practice.

The problems that remain are largely methodological and organizational. Good object-oriented systems analysis is still the key to the methodological aspects in my view. The real nightmare is the organizational issue.

All change management exercises are fraught with peril and the move to object technology is no exception. People often resist change merely because they doubt the benefits will actually accrue to them or their organizations. As Machiavelli (1961) put it: 'there is nothing more difficult to arrange, more doubtful of success, and more dangerous to carry through than initiating changes'. Those who prospered under the old order and stand to lose will oppose you vigorously, he says, while those who might gain will only provide lukewarm support. The potential gainers are 'generally incredulous, never really trusting new things unless they have tested them by experience'. To win them over one must demonstrate success early on. Furthermore, one must be sure of not backing the wrong horse. This is the danger of the silver bullet. I wonder how many hours of people's lives have been wasted backing a good technical solution that never survived commercially: the Xerox Sigma, APL, CP/M 86 and perhaps now even the Mac. I worked once in an IT organization that traversed the road to open distributed systems successfully. We achieved this by a positive turn to object technology, by strong leadership and clear vision and by hedging our bets on technology. I do not intend to write a history of that organization's experience. That would take too long and be too specific to be of general use. What I can offer is a distillation of advice.

There is no point in dithering. The aim of the migration strategy should be clearly stated in the form of a mission statement. It has to come from the top. People will accept it or vote with their feet in that case. One might announce that the mainframes are to be replaced within two years or that productivity will be doubled in three; ambitious targets which may not be fully met but which provide direction. These aims should be related to technology so that one might argue that reuse, using objects, is the basis for productivity growth. Next one must select some

technology. It is doubtful that one can be sure, in a new field, of what will survive or what will work. Therefore, rather than choosing just one language or one OOA/D method one should choose two or three, giving project managers some leeway to optimize their own performance. Typically the language choice should encompass different programming styles; so that a selection of C++ and Ada would be quite wrong. Different types of project can now be run with what appears to be the most suitable language and lessons learnt by the organization. Similarly, one may give project managers the authority to select from a short-list of object-oriented design methods. The analysis method, however, should not be treated in this way for, if it is, the reuse programme may be endangered. What is required is a sound method that imposes a definite life cycle and guides requirements capture, systems analysis and logical design. In this way, objects can be captured and placed in a repository from the earliest stages of a project. If the method is language independent these objects can be reused regardless of the language used. Think of the dismay that might be caused if a switch of language meant that the reuse library had to be ditched or even substantially redesigned. For those who would answer that wise companies must choose the industry standard UML notation (see Chapter 6) I would agree but remind them that a notation is **not** a method.

The development staff are critical too. The main choice that most organizations will face is that between traditional developers with many years in the business and great experience and newcomers with skills in OT and modern methods. In fact, neither type will do. The newcomers must be used to educate the oldsters and provide mentoring. Obversely, the young Turks must diligently learn about the nature of the business from their predecessors. It is a process that requires a deep and lasting trust between the two groups and establishing such trust is a key management task. Reward schemes that encourage both co-operation among workers and the reuse of other people's objects need to be devised.

Finally, the sponsors of computer developments must be rewarded too. The obvious rewards that can be aimed for are lower costs through reuse and shorter times to market. The quality of the product is thereby enhanced as well. Inheritance means that maintenance cost can be reduced and systems changed more easily in response to business change. Reuse successes lead to funding but are hard to achieve. Most developers expect to get reuse at the code level but this is very difficult to manage and often takes place informally. I believe that a reuse programme should start at the specification level .

4.5 Summary

Object technology makes the sharing by different systems possible. This chapter emphasized the rôle of object-oriented models as a way of describing and understanding distributed systems.

The operating systems of distributed systems can be *distributed* or *networked*. There are three data management strategies for distributed computing: centralized, replicated and partitioned. We examined several kinds of distributed computer architecture: client-server, multi-client/multi-server and peer-to-peer. We distinguished four distribution models: database server, transaction server, peer-to-peer and distributed front-end. The peer-to-peer model is the most general but the most difficult to program and manage. Latency is an important concept in the design of any object-oriented system, distributed or not.

Existing structured methods contribute little and impede progress with distributed systems. Object technology provides a natural way of modelling distributed applications. Network nodes can be regarded as abstract data types or objects. However, inheritance and composition links shouldn't cross the network. Associations that span the network should be minimized for reasons of efficiency. Typically, a system layer should be implemented at a single node. The object-oriented model both assumes and implies locational transparency.

Client-server computing was defined as the division of processing and data between one or more front-end client machines that run applications and a single back-end server machine that provides a service to each client. Client-server computing is a special case of distributed computing. Object-oriented systems are nothing other than, possibly layered, multi-client/multi-server or peer-to-peer systems. N-tier models are more flexible than their 2-tier antecedents. RPCs violate locational transparency but this is sometimes hidden from the user by clever network software. Co-operative processing is a special case of distributed computing that offers peer-to-peer communication between servers.

Database tools are often organized around the client-server metaphor. With relational systems, the ones that make use of stored procedures are able to reduce network traffic considerably. The difficulty is that encapsulation is not enforced and the discipline of a robust approach to object-oriented analysis is required. Object-oriented systems give these benefits automatically.

ORBs and other kinds of middleware remove much of the need for complex RPCs. Applications need not be written in an object-oriented manner since the ORB effectively provides a wrapper. ORBs bring the benefits of OT to the world of systems integration. XML is needed to define the meaning of the data included in message, but is not enough on its own to ensure that disparate systems work well together; for that a common business model is needed too.

The key technical features in this environment are distribution, connectivity and flexibility. These factors have led away from the waterfall-structured-mainframe mind-set to a rapid development, evolutionary, downsized culture. Advanced, object-oriented programming environments are essential to exploit prototyping and describe the parallel, distributed, message based world. Object-oriented analysis and design approaches are needed. Without sound object-oriented analysis and a project management culture that accepts its implications the distributed solution will soon become even more cumbersome to manage than the mainframes it replaced.

We examined various strategies for organizations facing migration and inter-

object-oriented implementation, reusing its components, extending it or building a better or a distributed front-end.

When the old system exists and is maintained in multiple versions, the wrapper approach will only work if there is a core system common to all the versions. Use a phased approach that builds a wrapper to communicate with new object-oriented components using (most probably) the borrowing strategy. Perform an object-oriented analysis on the old system. Use translation or data-centred translation to migrate.

Having decided to build a wrapper and a new front-end one needs tools for building them. There are no specific products offering wrapper technology for migration at present but there are a number of ORB and GUI tools that may help with reuse. Rather than build a wrapper for each old system it is sometimes better to wrap the communication system in some way. One approach to this problem is to use an object request broker. Many windowing environments now include good, usable GUI class libraries and facilities for developing object wrappers, so that existing, conventional UI code can be utilized. Sometimes the main tool can be the object-oriented analysis approach itself but good CASE tools are then required.

modelling computer networks. Comer (1999) discusses distributed computing at a more technical level, including issues such as network protocols and how RPCs work.

Agha, *et al*

7. What is the difference between CORBA and MOM?

8. Describe a CORBA product in detail.

9. What is the name given to the OMG standard for inter-object communication?
   a) Object Services Architecture
   b) Common Object Request Broker Architecture
   c) Object Interface Definition Standard
   d) None of the above

10. Define the terms: stub, skeleton, proxy, IDL and marshalling.

11. Discuss and contrast the use of XML in e-commerce and one other area such as publishing.

12. Why is XML and middleware not enough to ensure successful Enterprise Application Integration ? Discuss the problems to be expected on a typical EAI project.

13. Enumerate the advantages of XML in contrast with other approaches to EAI.

14. What does the S in XSL stand for?

15. Which concept is said to help with migration to object technology but safeguards investment in existing code?
   a) Coupling
   b) Object wrapper
   c) Overloading
   d) Cohesion
   e) Polymorphism

16. What is an object wrapper? How could one be implemented?

17. *'Object wrappers are pure idealism. The idea will never work in practice'*. Discuss.