Chapter 13 The OMG way: CORBA, CCM, OMA, and MDA

The Object Management Group (OMG), founded in 1989, is by far the largest consortium in the computing industry. OMG operates as a non-profit organization aiming at the standardization of 'whatever it takes' to achieve interoperability on all levels of an open market for 'objects'. Early 2002, around 800 member companies had joined OMG.

13.1. At the heart - the object request broker

Originally, OMG efforts concentrated on solving one fundamental problem: how can distributed object-oriented systems implemented in different languages and running on different platforms interact? Far from the problems of distributed computing, such simple phenomena as total incommunicando between code generated by two C++ compilers on the same platform stopped integration efforts right at the start. Differing object models from language to language made this worse. Differences between platforms coupled by low-level socket communication or - in better cases - by remote procedure call (RPC) packages completed the picture of deep gaps everywhere. The first years of OMG went into tackling these basic 'wiring' problems. The outcome was the Common Object Request Broker Architecture (CORBA) in its initial version, 1.1, released in 1991, followed by minor improvements in version 1.2. Today's highly successful standard is CORBA 2, version 2.0, released in July 1995 and updated in July 1996 (OMG, 1997a). The current version is 2.6, released in December 2001. The CORBA 3 specification is, as of early 2002, largely finalized but still pending. While numerous contributions originally scoped in CORBA 3 have been released in successive CORBA 2 increments, the biggest contribution, the CORBA Component Model (CCM), remains CORBA 3 proper and pending.

From the beginning, the goal behind CORBA was to enable open interconnection of a wide variety of languages, implementations, and platforms. Thus, OMG never settled on "binary" standards (standards at the level of deployable executables) - everything is carefully standardized to allow for many different implementations and for individual vendors of CORBA-compliant products to add value. The downside of this very open approach is that individual CORBA-compliant products cannot interoperate on an efficient binary level, but must engage instead in costly high-level protocols. The most prominent, although only moderately efficient, interoperability protocol is OMG's Internet inter-ORB protocol (IIOP), standardized with CORBA 2.0 in July 1995. Any ORB claiming interoperability compliance has to support IIOP. In the July 1996 update of the CORBA 2.0 standard, an Interworking standard was added, which specifies the interworking of CORBA-based systems with systems based on Microsoft's COM (see Chapter 15).

![Figure 13.1 Simplified structure of an ORB-based system.](image-url)

CORBA essentially has three parts: a set of invocation interfaces, the object request broker (ORB), and a set of object adapters. Invocations of object-oriented operations - also called...
method invocations - require late binding of the implementation. The method implementing
the invoked operation is selected based on the object implementation to which the receiving
object's reference refers. Invocation interfaces enable various degrees of late binding. They
also marshal an invocation's arguments such that the ORB core can locate the receiver object
and the invoked method and transport the arguments. At the receiving end, an object adapter
unmarshals the arguments and invokes the requested method on the receiver object. Figure
13.1. illustrates the basic CORBA structure in simplified form.

For invocation interfaces and object adapters to work, two essential requirements
need to be met. First, all object interfaces need to be described in a common language.
Secondly, all languages used must have bindings to the common language. The first condition
enables construction of generic marshaling and unmarshaling mechanisms. The second allows
calls from or to a particular language to be related to the common language. This common
language formed an essential part of CORBA from the beginning and is called OMG interface
definition language (OMG IDL). Here is an example of an OMG IDL specification:

```idl
module Example {
    struct Date {
        unsigned short Day;
        unsigned short Month;
        unsigned short Year;
    }
    interface Ufo {
        readonly attribute unsigned long ID;
        readonly attribute string Name;
        readonly attribute Date FirstContact;
        unsigned long Contacts();
        void RegisterContact(Date dateOfContact);
    }
}
```

Bindings to OMG IDL are available for several languages, including C, C++, Smalltalk, PL/I,
COBOL, and Java. Once interfaces are expressed in OMG IDL, they can be compiled using
an OMG IDL compiler and deposited in an interface repository, which every ORB must have.
By means of the ORB interface, compiled interfaces can be retrieved from the interface
repository. Also, when compiling program fragments that can provide implementations of
such interfaces, these program fragments, called object servants, can be registered with the
ORB's implementation repository. An ORB is capable of loading and starting an object
servant when receiving invocation requests for an object of that servant. An object adapter is
responsible for telling an ORB which new object is served by which servant. Multiple
servants can execute in a shared server environment, typically a process.

To enable efficient marshaling and unmarshaling of arguments, an ORB-specific
OMG IDL compiler must be used to generate stubs and skeletons. A stub can be instantiated
and then looks like a local object, but forwards all invocations through the ORB to the real
target object. In other approaches, stubs are called (client-side) proxy objects. A skeleton
receives invocations, unmarshals arguments, and directly invokes the target method. Although
not mentioned so far, a skeleton also accepts return values, marshals these, and sends them
back to the stub for unmarshaling and final returning. In other approaches, skeletons are
called (server-side) stubs.

Stubs and skeletons are good solutions when dealing with regular method
invocations. However, sometimes this binding is too static and the operation to be invoked
needs to be selected at runtime. CORBA provides a dynamic invocation interface (DII) for
this purpose, while CORBA 2.0 added a dynamic skeleton interface (DSI). These interfaces
allow for the dynamic selection of methods either at the client's end (DII) or at the server's
end (DSI). Both interfaces use a universal data structure for arguments to cater for methods of
arbitrary signature. Older versions of IONA's Orbix, for example, generated stubs that
translate static invocations to non-locat objects into sequences of DII calls. These Orbix
ORBs then handled only the universal dynamic invocation structures. Significant performance improvements can be gained by ORBs that implement DII/DSI directly and, since the introduction of IIOP, this is now commonly done. Figure 13.2 gives a more detailed view of CORBA and its interaction with the OMG IDL.

**Figure 13.2** CORBA and OMG IDL.

It is important to understand that the separation into calling client and called object does not impose an asymmetric architecture, such as client-server computing. The same process can be both issuing and receiving calls. Distribution of functionality to machines is left to the system's architect using CORBA. The object adapter introduces the only asymmetry. Programs that need to function as object servants need to register with the ORB via the object adapter. In theory, there can be different object adapters for the same ORB and these could even be used concurrently to serve different kinds of objects. Originally, OMG standardized the basic object adapter (BOA). In 1998, the BOA specification was deprecated and replaced by the specification of the portable object adapter (POA). The main problem with BOA was its undespecification that forced vendors to provide proprietary extensions, eliminating any hope for porting CORBA object implementations from one ORB to another.

Once an object servant is registered with an ORB, the ORB 'knows' how to activate that servant when needed. To determine on which machine to activate the servant, each registered object has a home machine that is used to start the servant on. Pure application programs that only call objects (but do not export any of their own), do not register with an ORB and therefore cannot be started by an ORB.

The OMG IDL distinguishes between basic and constructed data types and CORBA object references. Data types include integers, floats, characters, strings, structures, sequences, and multidimensional fixed-size arrays. All data types are passed by value. Before CORBA 2.3, CORBA objects themselves could not be passed, only references to CORBA objects could. Starting with CORBA 2.3, objects can also be passed by value and a standard mapping of such values to XML is defined. CORBA object references are opaque types and different from the references used within a bound language; they are much larger and cost more than the native references. Again starting with CORBA 2.3, there is a standard way to form an object reference as a URL (as in `corbaloc::www.example.org/CoolService`).

The ORB interface provides operations to turn a native reference into a CORBA reference and back. It also provides operations to turn a CORBA reference into a unique but proprietary ORB-specific string and back. Such strings can be used to store CORBA references – and are typically used within IIOP exchanges. A CORBA reference is defined to have indefinite lifetime – that is it will never be reused. The attempt to retrieve the associated object may of course fail if that object has been deleted in the meantime.

Almost all CORBA standards are, in the end, specific over interfaces defined using OMG IDL. With CORBA 3.0, OMG is moving beyond IDL by introducing two new
languages that focus on class rather than interface properties. The first is the persistent state
definition language (PSDL) that primarily captures storage types (records with typed fields) and storage homes (factories) - for more information see the discussion of the persistent state service in section 13.2.2. The second new language is the component implementation
definition language (CIDL), itself an extension of PSDL, adding components, component homes, composition entities, composition processes, and executors (for details see section 13.3).

13.1.1. From CORBA to OMA

CORBA 2-compliant ORB implementations are available from several vendors on many
platforms, but CORBA 3-compliant implementations were slow to materialize (as of early
2002). The above discussion should have made clear that an ORB is essentially a remote
method invocation service. As such, ORBs promise a much cleaner model to program
distributed systems than services based on remote procedure calls or even lower-level
abstractions. Indeed, the most common use of ORBs in industry is to replace sockets and
remote procedure calls in applications spanning several server machines. The pure 'wiring'
standard established with CORBA is thus successful. However, above this basic 'wiring',
programmers were still left alone before CORBA 3. Although the communicating ends may
be on different machines and implemented in different languages, they need to share many
conventions to interoperate. As a result, the ends are still most likely to be developed by the
same team.

Being aware of this shortcoming, OMG started to broaden its focus long ago. Since
CORBA 2, the OMG's overall effort is called the object management architecture (OMA)
(OMG, 1997b). Today, it revolves around the CORBA 3 specification, including OMG IDL,
language bindings, invocation interfaces, object adapters, interface and implementation
repositories, object servants, and component infrastructures. The OMA adds three new areas
of standardization, which are a set of common object service specifications (CORBAservices), a set of common facility specifications (CORBAfacilities), a set of
application object specifications, and, since CORBA 3, the CORBA Component Model
(CCM), also called CORBA components. Figure 13.3 presents an overview of the OMA.

![Diagram](image)

**Figure 13.3** The OMG's object management architecture (OMA).

Object services support all CORBA-based programs in a way that is independent of specific
domains or application models. Object services concentrate on the fundamental building
blocks of any distributed solution, such as event propagation, transactions, or naming.
Common facilities are either horizontal or domain-specific. Horizontal facilities are domain-
independent, but focus on specific application models - these are increasingly de-emphasized
and play almost no role in CORBA 3. For example, the once prominent (and most complex)
standardized horizontal facility - OpenDoc, supporting compound documents - is now effectively defunct. Vertical facilities with their focus on specific domains, on the other hand, had a slow start but now are growing strongly.

Finally, application objects add domain-specific entities that could be plugged into component frameworks. The most prominent class of application objects is business objects. These are objects that directly represent abstractions used in specific businesses. Today, this category is essentially void. No specifications of application objects are presently part of OMA.

13.1.2. CORBA timeline

The following list places the CORBA versions on a timeline (unexpanded acronyms are explained elsewhere in this book - see glossary and index):

- CORBA 1.0 - October 1991
  - IDL, DII, Interface Repository, C mapping
- CORBA 1.1 - February 1992
  - BOA and memory management; clarified Interface Repository as well as object model.
- CORBA 1.2 - December 1993
  - Several clarifications, especially in memory management and object reference comparison.
- CORBA 2.0 - February 1997
  - DSI, initial reference resolver for client portability, extensions to Interface Repository, interoperability architecture (GIOP, IIOP, DCE CIOP), layered security and transaction services, data type extensions for COBOL, scientific processing, wide characters, COM and OLE Automation interworking, language mappings for C++ and Smalltalk.
- CORBA 2.1 - September 1997
  - Secure IIOP, IIOP over SSL, language mappings for COBOL and Ada, interoperability revisions and IDL type extensions.
- CORBA 2.2 - July 1998
  - Server portability enhancements (POA), DCOM interworking, language mappings for Java, reverse mapping Java to IDL.
- CORBA 2.3 - December 1998
  - Revisions of: language mappings for C/C++, Java, ORB portability, COM and automation interworking, CORBA Core, ORB interoperability, and security.
- CORBA 2.3.1 - October 1999
- CORBA 2.4 - October 2000
  - Several quality of service (QoS) specifications. Contains asynchronous messaging, Minimum CORBA, and real-time CORBA specifications.
- CORBA 2.4.1 - November 2000
- CORBA 2.4.2 - February 2001
- CORBA 2.5 - September 2001
  - Fault-tolerant CORBA, portable interceptors.
- CORBA 2.6 - December 2001
  - New secure interoperation specification and general interoperability improvements.
- CORBA 3.0 - pending
  - Java and internet integration (objects passable by value and XML mappings, allow Java RMI objects to interoperate over the network as CORBA objects, standard URL protocol corbaloc for non-binary object references, firewall specification); quality of service control, real-time and fault-tolerant CORBA,
CORBA 1 was all about object request brokers and IDL is its hallmark contribution. CORBA 2 focuses on interoperation (and interworking) and IIOP is its hallmark. CORBA 3 aims to focus on component and system integration and CCM is its hallmark. This timeline exposes a fairly steady process, even if often much slower than initially hoped. For instance, CORBA 3 was originally scheduled to be finalized by late 1998.

In early 2002, the CORBA 3.0 specification was still not formally released. However, as they have been formalized one by one, several of the items targeted for CORBA 3.0 have been incorporated into CORBA 2.3 to 2.6 releases. In particular, objects passable by value, XML mappings (see section 13.7), and Java RMI interoperation were incorporated in CORBA 2.3, which marked the beginning of the mutual Java-CORBA co-evolution. Object reference URLs, asynchronous messaging, minimum, and real-time CORBA are part of CORBA 2.4. CORBA 2.5 added fault tolerance and portable interceptors; CORBA 2.6 secure interoperation. A Python mapping has been formalized (Python is a popular scripting language); IDLscript is pending. The main contribution of CORBA 3.0, the component model, was essentially complete by the end of 2001.

IBM's System Object Model (SOM) was deprecated in 1998 (the SOM 3.0 distributions for AIX, OS/2, and Windows NT are available as freeware from www.ibm.com/software/ad/som). The following brief discussion is kept as there are several historic references to SOM throughout this book.

SOM was originally developed independently from CORBA as part of the OS/2 workplace shell. Later, it was made first CORBA 1.2- and then CORBA 2-compliant. In fact, distributed computing is supported by the distributed SOM (DSOM) libraries, which build on SOM. In this section, DSOM is considered to be an integral part of SOM. SOM implemented a superset of the CORBA 2 standard and supported metaservices that are still not on the CORBA map. In addition, SOM defined a binary standard.

Two features of SOM stand out - its support for metaprogramming and support or binary compatibility across binary releases. The SOM metaprogramming model largely follows the Smalltalk example (Goldberg and Robson, 1993), so every class is itself an object and as such an instance of a metaclass. All metaclasses are instances of a single class, 

Meta\text{class}, which is its own metaclass. SOM goes beyond the reflective capabilities of CORBA as SOM allows classes to be constructed or modified dynamically. For example, it is possible to add a new method to an existing class without disturbing any of the existing instances of that class - these existing instances will immediately support the new method. There is at present no other mainstream component platform that supports a similar level of metaprogramming. Runtime code synthesis is supported elsewhere (CLR, Java), but these do not support modifications that affect already existing instances.

Versioning and binary compatibility are supported by the notion of a release order (Forman et al., 1995). For example, adding new methods to a later release does not alter the dispatch indices used by code compiled against an older release. SOM comes with precise rules as to which changes in a release maintain, and which other changes break, binary compatibility with previous releases. Binary compatibility is a very important issue in a component world. It is unthinkable to ask all vendors of dependent components - and the vendors of components dependent of these components, and so on - to recompile and redistribute within any reasonable time. This is the syntactic fragile base class (FBC) problem (section 7.4.1). JVM and CLR incorporate similar notions of release-to-release binary compatibility.

SOM guarantees binary compatibility across a large number of base class changes, including refactoring of class hierarchies, as long as the required methods remain available and of compatible signature. As a special case, SOM guarantees that, if no interface changes
took place, then building the next release of a component is guaranteed to preserve binary compatibility with clients compiled against the previous release. This effectively solves the syntactic FBC problem, but obviously cannot address the semantic FBC problem. (Tackling the semantic FBC problem requires full support for side-by-side existence of multiple versions of interfaces and implementations, as originally established by COM and fully supported by CLR. See Chapter 15).

13.2. Common object service specifications (CORBA services)

CORBA services currently specifies 16 object services, one of which (the notification service) is formally part of the telecommunications domain facility. The following sections present brief summaries of these services, in two categories. Per category, the order follows the much more detailed discussion by Siegel (2000). The first category covers the services relevant for today’s enterprise computing applications using CORBA. These applications typically use CORBA objects as modules and CORBA as a convenient communication middleware. The relevant services are those that support large-scale operations. The second category covers the services aiming at finer-grained use of objects. These are today seen as being of lesser practical importance. One potential exception with a particularly turbulent history could be the persistent state service (PSS) of CORBA 3 that replaced the CORBA 2 persistent object service (POS). While yet to be adopted by products, PSS is one of the three pillar services underlying the CORBA Component Model (described in section 13.3); the other two are the object transaction and the notification service (described in the next section). It is noteworthy that large CORBA-based systems frequently use only a few CORBA services, with the exception of naming and some sort of security and, to a lesser degree, transactions and trading. This reality is reflected in available ORB products, which tend to not support the full spectrum of CORBA services.

13.2.1. Services supporting enterprise distributed computing

Many large enterprise systems use CORBA essentially as an object bus, relying on ORBs to attach a wide variety of systems. Naming is the one key service that all such systems build on.

**Naming service, trader service**

Objects always have a unique ID used internally. The naming service also allows arbitrary names to be associated with an object. Names are unique within a naming context and naming contexts form a hierarchy. The resulting naming tree is quite similar to directory structures in file systems.

The trader service allows providers to announce their services by registering offers. Clients can use a trader to locate services by description. A trader organizes services into trading contexts. Clients can search for services, based on parts of descriptions and keywords, within selected contexts. The trader returns a list of offers that match the query. The OMG trader service specification was simultaneously adopted by the ISO (ISO/IEC 13235-1) and the ITU (ITU-T recommendation X.950).

The naming service can be compared to White Pages and the trader service to Yellow Pages.

**Event service, notification service**

This service allows event objects that can be sent from event suppliers to event consumers to be defined. Event objects are immutable in that information flows strictly in one direction, from supplier to consumer. Events travel through event channels that decouple supplier from consumer. Events can be typed (described using OMG IDL) and channels can be used to filter events according to their type.
The event channel supports both the 'push' and the 'pull' model of event notification. In the 'push' model, the event supplier calls a push method on the event channel, which reacts by calling the push method of all registered consumers. In the 'pull' model, the consumer calls the pull method of the event channel, effectively polling the channel for events. The channel then calls a pull method on the registered suppliers and returns an event object if it finds a supplier that returns an event object.

In 1998, the specification of the notification service added several critical features to the event service (Siegel, 2000) - quality of service (QoS) specification and administration; standards for typed and structured events; dynamic event filtering based on type and QoS; filtering at source, channel, consumer group, and individual consumer level; and event discovery among source, channel, and client. Note that, technically, the notification service is not a CORBA service but a CORBA facility introduced as part of the efforts of the telecommunications domain taskforce (TelDTF).

Object transaction service

The object transaction service (OTS) is one of the most important services to build distributed applications. The OTS was standardized by OMG in December 1994, and now forms part of most ORBs and several J2EE servers. An OTS implementation must support flat, and optionally can support nested, transactions. It is possible to integrate non-CORBA transactions that comply with X/Open distributed transaction processing standard. Integration with transactions spanning multiple and heterogeneous ORBs is also possible.

In the context of component-based systems, nested transactions seem unavoidable. It should be possible for a component implementation to create a transactional closure for a sequence of operations without having to declare this in the component interfaces. The principle of independent extensibility then requires support for nested transactions. Flat transactions, the only ones guaranteed to be supported in a compliant OTS implementation, are of limited value in a component system. However, this is a common shortcoming as practically no mainstream transactional systems today support nested transactions.

The OTS automatically maintains a current transaction context that is propagated along with all ORB-mediated requests and passed on to non-ORB transactional activities. For CORBA objects, the context is passed to any object that implements interface TransactionalObject. The current context can be requested from the ORB and thus is always available. The transaction operations begin, commit, and rollback are defined on the current context.

All objects that are modified under a transaction and require transactional control register with the OTS coordinator object. The relevant coordinator can be retrieved from the current context. A resource can indicate that it understands nested transactions. Resources have to implement interface Resource, which is used by the coordinator to run a two-phase commit protocol. (It is known that two-phase commit may deadlock in a fully distributed implementation unless it can be built on a specific kind of broadcast protocol; three-phase commit protocols are known to avoid this problem, but at a higher cost per transaction (Mullender, 1993). The OTS approach requires the coordinator to be logically centralized).

Instead of providing transaction control as a separate service, as promoted by the OTS design, it is now most popular to integrate transaction and other services into a context or container abstraction provided by an application server. (The CORBA model to do so is the new component model (CCM) described in section 13.3).

Security service

A robust security service is clearly of paramount importance for distributed systems spanning more than a single trusted organizational domain. The security service needs to be pervasive. All interoperating ORBs, and other interworking systems, need to collaborate, and a security policy needs to be established for all involved organizational units.
The CORBA security specification defines a number of services for tasks such as authentication, secure communication, delegation of credentials (also known as impersonation), and non-repudiation. Very few products actually support the full spectrum of security services in this specification. A few, such as BEA's WebLogic Enterprise and IBM's WebSphere go quite far. Many other products simply rely on Netscape's secure sockets layer (SSL) - especially in the case of standalone ORBs as compared to fully integrated application servers. Using SSL is fine to establish simple authentication and secure communication properties. However, it does not support more advanced concepts such as delegation or non-repudiation.

13.2.2. Services supporting architectures using fine-grained objects

Out of the following list of services, several have not made it into products, including the collection, externalization, and query services. The reasons range from loose specification, as in the case of the query service, to impractical assumptions, as in the case of collection service (Siegel, 2000). The following is, nevertheless, a complete list and brief description of all object services presently covered by the OMA.

**Concurrency control service**

This service supports acquisition and release of locks on resources. Locks can be acquired either within a transactional context (see object transaction service above) or within a non-transactional context. Locks acquired on behalf on a transaction will be released as part of a transaction's rollback. Locks can be acquired in one of several lock models, such as read, write, and upgrade. A read lock allows for multiple readers whereas a write lock ensures single writers. An upgrade lock is a read lock that can be upgraded to a write lock because it guarantees mutually exclusive read access.

Locks are acquired out of locksets. Each protected resource holds a lockset that determines what kind of locks and how many of them are available. A lockset factory interface supports creation of new locksets. Locksets are either transactional or non-transactional and can be related to other locksets. A lock coordinator object can be used to release all locks held in related locksets.

**Licensing service**

As soon as components are used to assemble solutions, there needs to be a way to obtain licenses for all but freeware components. The licensing service supports a variety of different licensing models. The service defines just two interfaces (abstractions): license service manager and producer-specific license service. If an object is bound by a license agreement it can itself use the license service manager to find out whether its use is legitimate.

A licensed object contacts the license service manager and obtains a reference to a producer-specific license service object. All further activities are with this specific object. The licensed object informs the specific service object that its use has started and passes information such as the component name and version, the object reference, and a user context. The specific service object checks whether or not a valid license exists for this user context and advises the licensed object about actions to be taken. For example, the licensed object may switch to demo mode or offer a grace period if no valid license exists or if the license has expired. The actual licensing policy is thus fully encapsulated by the licensed object and the producer-specific license service object.

Once operating, the producer-specific license service object periodically sends event notifications to the licensed object, which replies by reporting usage statistics. Alternatively, the licensed object could actively report at regular intervals. The reports can be used to maintain a usage profile or to implement license expiration policies. Finally, if the user stops using the licensed object, it informs the specific service object, which then stops sending events.
**Lifecycle service**

This service supports creation, copying, moving, and deletion of objects and related groups of objects. Containment and reference relations used to handle groups of objects are described using the relationships service outlined below. Where containment relations are used, copies are deep – all contained objects are also copied. To support object creation, the lifecycle service supports registry and retrieval of factory objects. Once the needed factory object has been retrieved, it can be used to create new objects.

Surprisingly, the lifecycle service offers a destroy operation to get rid of objects or groups of objects but does not help to determine when to destroy objects. This is a significant shortcoming of CORBA as subtle distributed memory management issues are simply left for higher levels to solve. By comparison, DCOM supports distributed reference counting and Java and CLR even support a form of distributed garbage collection based on remote reference leases that expire unless used or renewed.

In current enterprise applications built using CORBA, this is not normally an issue for several reasons. For one, certain ‘objects’ are of unbounded lifetime, and simply represent a traditional server program. In such a setting, CORBA is used as a communication middleware for modules distributed across a networked environment. Also, to be fair, distributed reference counting or garbage collection works well as long as there are no network or machine failures. To solve the distributed memory management problem in the presence of such failures requires embedding in a transaction context. Long-lived transactions are necessary to manage properly the lifetime of longer-lived objects in a proper fashion and long-lived transactions do not form part of mainstream offerings. Therefore, a different approach is commonly used. This involves building of short-lived transactions and keeping all relevant state in transacted stable stores (databases). The lifetime of objects is then either delineated by the lifetime of transactions or by that of database entries. In the latter case, the object itself is merely a reconstructable cache of the database state.

**Relationship service**

The relationship service was meant to allow general relationships between objects to be specified and maintained. Rather than resorting to language-level pointers or references, this service introduces an associative model that allows relationships over objects to be created without changing the involved objects at all. However, the relationship service is rarely used or even implemented and is likely to be replaced by support for relationships among business objects based on CCM (Siegel, 2000).

**Persistent state service**

Object persistence is the property of an object to survive the termination of the program that created it. Starting with CORBA 2, the persistent object service (POS) was meant to support persistence of CORBA objects. Despite being a key service standardized by OMG in early 1994, it took implementations until mid-1996 to appear in the first beta releases. Some reports on implementation attempts even pointed out severe technical problems with the specification and its expected interoperation with other object services, such as that with the object relationship service (Kleindienst et al, 1996). In addition, it emerged that the POS did not solve the ‘right’ problems. In particular, it was still left to application code to request object storage. The POS specification was finally deprecated and replaced in CORBA 3 by the new persistent state service (PSS).

The fundamental idea behind any persistence service is to provide an abstraction layer that shields persistent objects from the persistence mechanism. For example, objects can be stored in files, in relational or object databases, or in structured storage as used by compound document architectures. There are two basic operations, which are storing an object and retrieving an object. However, three properties of objects make these operations a non-trivial undertaking.
First, objects have an observable identity - that is, they are not referentially transparent. A persistence service must thus ensure that object identity is preserved. If an object that has been stored before is stored, then the original copy is updated and a reference to that is stored. Likewise, if an object that has been retrieved before is retrieved, and is still reachable, then a reference to the previously retrieved object is returned.

Second, objects refer to each other and thus form an object web. These references need to be maintained across persistent storage of objects in a web. It must be possible to distinguish between essential and transitory object references; otherwise a large number of temporary objects would be dragged into the persistent store. Also, if multiple persistent stores are used, relations must keep across such stores. In addition to programming language level references, a POS must also support relations introduced using the object relationships service.

Third, objects are units of encapsulation. Despite their storage in persistent stores, an object's contents should be protected against direct manipulation, by-passing the object's encapsulation barrier. Of course, this level of protection is only feasible with certain persistent stores.

The original POS aimed to resolve these issues by means of protocols between persistent objects and persistent data services. As these protocols require the cooperation of the involved objects with the persistence service, they were sometimes called conspiracies (Orfali et al., 1996). The resulting design would have led to an entangling of objects to be persisted and the various stores - probably one reason for the failure of the POS attempt.

With the new PSS, the approach to determining what to store for a given object type has been made explicit and declarative. Either a programming language, such as Java, is used that has built-in support for persistency declarations on fields of objects, or a separate persistence declaration is written using the OMG persistent state definition language (PSDL). Essentially, declarations in either form establish a schema for the storage representation, split into abstract storage types. This model is quite close to the distinction between abstract and concrete classes, or interfaces and classes. Instead on focusing on the abstraction of operations, PSDL focuses on the abstraction of persistable state. In addition PSDL supports abstract and concrete factories (called storage homes) for instances of given storage types.

**Externalization service**

This service supports mapping of an object web to a stream and back. The process of first externalizing the objects and then internalizing them again creates a copy of the corresponding object web. The externalization service does not maintain referential integrity. It merely preserves the references between objects externalized together. Externalization can thus be used to copy object webs by value. References to other objects can be maintained explicitly using ORB provided string identifiers for these references.

To become externalizable, an object needs to implement the `Streamable` interface. Externalization if an object is requested by invoking an externalize method on an object implementing the `Stream` interface. This stream object invokes the `externalize_to_stream` method of the streamable object and passes an object implementing the `StreamIO` interface. The streamable object can then use this `streamIO` object to write any of the OMG IDL-defined data types or to write embedded objects. The streamable object can also externalize an entire graph of objects defined using the relationship service.

**Properties service**

This service allows arbitrary properties to be associated with objects which implement at least interface `PropertySet`. Properties can be added, retrieved, and deleted individually or in groups. If an object also implements interface `PropertySetDef`, properties can be further controlled to be of one of four property modes. Properties can thus be normal (can be modified or deleted), read-only (can be deleted but not modified), fixed-normal (can be modified but not deleted), or fixed-read-only.
The property service does not interpret any of the properties associated with an object. Properties are useful in programs that, generically, need to attach information to arbitrary objects. An important example is system administration tools that attach 'stickers' in order to track objects efficiently.

**Object query service**

This service helps to locate objects by attributes. It is similar to the object trader service, but instead of locating servers it locates object instances. Queries are based on the attributes that objects make public or accessible via operations. Two query languages are supported: the Object Database Management Group's ODMG-93 object query language (OQL) and SQL with object extensions. A single common query language is under development.

The query service defines its own simple collection service – a subset of the general collection service. Collections are used while processing queries to form result sets and are then returned to the querying client. These simple collections provide ordered set semantics, including operations to add and remove elements or sets of elements. The service also provides an **Iterator** interface to support enumeration of the elements of a collection.

The query service defines four query-related entities, each with its OMG IDL-defined interface: *query objects*, *query evaluators*, *query managers*, and *queryable collections*. A query object encapsulates the query itself and operates in two stages: first, the query is prepared and then the query is executed. A query evaluator can take a query and operate over a queryable collection to process the query and return a result: again a collection. A query manager creates query objects and delegates queries to the relevant query evaluators. The querying client finally uses an iterator to work through the collection of returned results.

**Object collections service**

The collection service supports collections of various abstract topologies, such as bags, sets, queues, lists, or trees. The role model is the Smalltalk collection classes library (Goldberg and Robson, 1983, 1989). It is debatable whether the CORBA collection service - based on the relatively heavyweight model of CORBA objects - can ever be competitive with native object collection libraries. At the same time, object databases may be better suited to transfer 'collections' of various shapes and with various properties across ORBs.

**Time service**

This service deals with the inaccuracies inherent in a distributed system with multiple asynchronous clocks. In many applications, realtime information is used to correlate internal events, such as creation of files, with universal time. A time service has to ensure that such correlation is possible within reasonable error margins and that non-causal correlation is avoided. As an example, consider the creation of a new object as a reaction to another object firing an event. Non-causal time-based information would result when assigning a 'date of birth' time stamp to a new object that predates the first object – a typical result of a non-causal time service.

**13.3. CORBA Component Model**

CORBA 3 is the latest incarnation of CORBA standards (Siegel, 2000). Although, as of mid 2002, the final set of specifications is still forthcoming, substantial strides have already been made to improve on almost all aspects of CORBA 2. Besides an overhaul of many object services, the single biggest contribution is probably the new CORBA Component Model (CCM) - although the release of the final CCM specification is itself still pending. (Occasionally, CCM is also referred to as CORBAcomponents).

CCM is an ambitious logical extension of Enterprise JavaBeans (see section 14.5.2). CCM introduces several novel features, promises a fully compatible embedding of existing
EJB solutions and aims to maintain the original CORBA tenet of being both language and platform independent.

It remains to be seen if CCM can break free from being a mere paper extension of EJB - that is, whether or not viable CCM-compliant products will emerge. Such products would go beyond the current tendency of J2EE application servers that largely use CORBA only as an interoperable connectivity standard via IIOP. Although the CCM specification was close to being finalized in mid 2001, only a few commercial implementations had been announced by then. Most notably, IONA committed to delivering a CORBA-3 compliant implementation almost in synchrony with the finalization of the specification.

The following subsections cover the most important CCM features, including the portable object adapter (POA) that is essential for several CCM features.

13.3.1. Portable object adapter

The main function of a CORBA object adapter is to mediate between an ORB and the actual implementation of an object receiving incoming calls and returning results. Replacing the now outdated basic object adapter, the current object adapter specification is that for portable object adapters. Presently, there is no other object adapter specification. An instance of the portable object adapter accepts requests on behalf of a group of objects. In any server process supported by an ORB there will be at least one POA instance; but there may be as many as one per serviced object in that process.

Clients direct calls to object references. Object references are created by POAs at the server side and then passed to clients. To enable clients to talk to any objects, a set of well-known objects is made available by ORBs on the client side, typically including an object providing the naming service. The object reference contains the information needed for a client ORB to locate the server ORB, which then receives the client's call. Furthermore, the object reference contains the name of the POA that created the object reference and an object ID that is valid only relative to that POA. A POA's name is assigned on initial creation in a server, while ORBs maintain a registry of such named POAs and can reactivate POAs if necessary.

Figure 13.4 Generation of stub, POA skeleton, and servant template from given IDL specs
A POA instance handles an incoming request by handing it off to a servant. Servants are the implementations of CORBA objects. Figure 13.4 shows a typical tooling scenario, where, starting from an IDL definition, a client-side stub, a server-side POA skeleton and a server-side servant template is generated. A developer can then fill in the implementation details by completing the generated template. A CORBA object is not necessarily implemented using object-oriented languages and servants are therefore not necessarily classes. If object-oriented languages are used, servants are instances of classes.

POAs create servants following a number of different possible policies. Objects, and with them their servants, can be activated explicitly by, for example, a server startup program. Explicit object activation yields the fastest response times but also has the highest impact on resources. Alternatively, a single servant can be used for all objects of a kind with either one servant per type or one servant for multiple types. In this case, the servant can implement resource management mechanisms to keep only active objects in memory. A servant itself can also be activated on demand by an ORB for either the duration of a single method call or for the lifetime of the hosting server process.

13.3.2. CCM Components

A CCM application is an assembly of CCM components, each of which may be custom-built or off-the-shelf, in-house or acquired. Enterprise JavaBeans (EJB; see section 14.5.2) components and CCM components can be combined in a simple application. (For readers unfamiliar with EJB, it may be preferable to read relevant parts of Chapter 14 before proceeding). Individual components are shipped in component packages that contain an XML document detailing their contents, which can include binaries for multiple platforms. CCM assemblies contain an XML document describing the set of component packages they refer to and the deployment configuration of these.

A CCM component itself can consist of multiple segments. CCM runtimes load applications at the granularity of segments (see Figure 13.5). As CCM requires special server-side support, CCM applications can only be executed on CORBA-3 compliant ORBs. Perhaps surprisingly, a CORBA 3 ORB is also required at the client end to achieve full CCM fidelity. CCM does support so-called component-unaware clients on pre-CORBA 3 ORBs, but such clients will not have access to several CCM features, such as navigation operations.
assign servants to component instances. Service components are instantiated per incoming call and thus cannot maintain state across calls (they correspond to EJB stateless session beans). Instances of service components maintain state for the duration of a transactional session and allow for multiple calls within such a session. Across session boundaries, instances of service components lose their state. Instances of process components have persistent state - their lifetime corresponds to the lifetime of some process they are servicing and is as such arbitrary. Entity components finally have persistent instances that correspond to entities in some database - they can be accessed by presenting the database entity's primary key.

A CCM component is programatically characterized by a number of features (Figure 13.6).

- Ports that are classified into facets, receptacles, event sources, and event sinks. A facet is a provided interface and a receptacle is a required interface. A component instance's receptacles are connected to other instances' facets. Event sources and sinks are similar, but instead of being connected to each other, they are both connected to event channels.
- Primary keys, which are values that instances of entity components provide to allow client identification of these instances.
- Attributes and configuration, which are named values exposed via accessors and mutators.
- Home interfaces, which provide factory functionality to create new instances.

A special facet (interface) of a CCM component is the equivalent interface, which enables navigation between the different facets of a CCM component. Each facet interface can be provided by a part object that is encapsulated by the CCM component's outer instance. The support of multiple facets and navigation via a joint equivalence is very similar to the COM design (Chapter 15) of aggregation and navigation based on the IUnknown interface. Clients require special ORB support to be able to navigate the facets of a CCM component instance. Such component-aware clients need to run on a CORBA 3 ORB.

![CCM Components with their many features and two styles of explicit connections](image.png)
Receptacles provide connect and disconnect operations and internally correspond to object references to other objects of appropriate type. Connections can be made declaratively in CCM deployment configurations or made or broken dynamically at runtime.

Configuration interfaces support initial configuration of new component instances. They are described as IDL attributes with set and get operations. (CORBA 3 extends IDL to allow for exceptions being thrown by attribute setters and getters). A special call signals completion of configurations as, before that, calls on operational interfaces are disallowed and, after that, calls on configuration interfaces are disallowed.

The home interface is provided by a component, not its instances, and supports the creation of new interfaces. Home interfaces also have other lifecycle-related operations for the objects they manage, such as lookup operations based on primary keys or destructor operations. A component provides at least one, but can provide multiple home interfaces.

### 13.3.3. CCM Containers

CORBA 3 defines a component implementation framework (CIF), which includes generators that accept CIDL (Component Implementation Description Language) input and generate implementation code that completes explicitly provided component code.

In addition, every component instance is placed inside a CCM container (Figure 13.7). Components interact with POA as well as transactions, security, persistence, and notification services via interfaces on their container. A container also has receptacles that accept callbacks into the component instance.

![Figure 13.7 CCM containers](image)

A number of options are available for each of the four services that CCM packages. Transaction control can be container-managed or self-managed. In the container-managed case, a component configuration states if transactions are supported, required, required new, or not supported. The container will begin and end transactions to meet these requests. Similarly, persistence can be declared as container-managed or self-managed. In the container-managed case, PSDL is used to declare what needs to be persisted. For security, required access permissions can be declared on operations in CIDL and will be checked by the container.

### 13.4. CORBA-compliant implementations

By early 2002, several CORBA services had not been adopted widely nor had they even been implemented seriously. Such service specifications may be retired in the near future. Also, most service implementations are bundled with ORBs and not sold as separate products, but there are exceptions. Moreover, most ORBs aren't sold as individual products either, but bundled in larger application server products, such as IBM's WebSphere, BEA's WebLogic Enterprise, IONA's Orbix E2A Application Server Platform, or Borland's Enterprise Server,
all of which combine a J2EE application server with CORBA ORB and service functionality. It is noteworthy that IBM's WebSphere and BEA's WebLogic are also the premier implementations of J2EE in the market (in early 2002). IONA continues to provide several stand-alone CORBA ORB products and specializes on reaching a broad spectrum of platforms, including mainframes.

On the CORBA 3 front, activities looked slow in early 2002. However, in May 2002, the Internet Component Management Group (iCMG) announced the availability of its K2 Component Server 1.2, a server-side infrastructure based on CCM (www.icmgworld.com/corp/k2/k2.overview.asp). Interworking with EJB was still at the level of EJB 1.1 - as was then still mandated by the CCM specification.

The following brief descriptions of some available CORBA-compliant products may help to gauge the reach of CORBA (at the time of writing). Beyond that, the descriptions are neither comprehensive nor is the list of products in any way complete.

13.4.1. BEA's WebLogic

The BEA WebLogic platform is a family of products including a J2EE application server (WebLogic server), an integration server (WebLogic Integration), and a web server (WebLogic Portal). A dedicated integrated development environment (WebLogic Workshop) enables creation, testing and deployment of applications and web services built using the WebLogic products. Under the umbrella WebLogic Enterprise Platform, BEA subsumes WebLogic products as well as its Tuxedo product. (Starting with version 8.0, Tuxedo includes the former WebLogic Enterprise 5.1 "T-engine", while all Java and EJB-related technology is exclusively located in WebLogic products).

The WebLogic Server integrates with BEA's Tuxedo transaction processing monitor via the WebLogic Tuxedo Connector at the levels of IIOP connection pooling, transactions, and security. As today's IIOP doesn't cover all requirements (especially in the area of security), WebLogic also supports BEA's proprietary T3S protocol. Tuxedo includes a CORBA 2-compliant ORB for C++ and Java, including implementations of object lifecycle, naming, notification, security, and transaction services. An interface repository is supported. Tuxedo supports C/C++ and Cobol applications based on procedural, CORBA and ATMI (application to transaction manager interface) programming models. Tuxedo also supports the CORBA specification for distributed application development. Finally, ActiveX clients can access Tuxedo ORB via COM automation interfaces.

Tuxedo supports load balancing of objects and requests across replicated server processes and server groups. Routing of requests can be data-driven (ATMI only) or factory-based (CORBA). Multiple incoming client connections can be multiplexed. Multithreaded services can be constructed. Tuxedo has a pluggable security framework that can be customized to integrate with external security systems. As of WebLogic Server 7.0 and Tuxedo 8.0, both products use the same pluggable security framework, easing their integration.

13.4.2. IBM's WebSphere

The WebSphere Applications Server family of products includes standard, advanced, and enterprise editions. (A number of IBM products are also branded WebSphere, but these are less relevant here). The standard edition is essentially a web server supporting Java servlets, Java server pages, and XML. The advanced edition adds J2EE application server capabilities by supporting EJB. The enterprise edition, finally, incorporates a CORBA 2-compliant ORB.

The WebSphere advanced edition's J2EE implementation stood out in early 2002 for its support for EJB 2.0 container-managed persistence and relationships in a way that integrates tightly with the DB2 caching system. This is important as container-managed persistence and relationships reduce an application programmer's control over performance characteristics by either manifesting too many entity instances (eager) or too few (lazy). Integrating with the database cache reduces the impact on performance.
WebSphere enterprise edition builds on IBM's ComponentBroker, an approach to packaging existing software into components and supporting simple set-up of connections between such components. In original ComponentBroker announcements, IBM stated that an "estimated 70 percent of all code written today consists of interfaces, protocols, and other procedures to establish linkage among various systems". ComponentBroker supports CORBA and Java to enable connectivity. In addition, ComponentBroker provides a system-wide management facility to control aspects such as security, adequate resourcing to satisfy demands, and proximity of location to improve performance.

ComponentBroker considers CBToolkit and CBConnector. The toolkit facilitates development of components whereas the connector enables connection and management of components. The ComponentBroker programming model is to separate business models from client views. Business objects encapsulate functionality according to business functions. Client views are visual objects. Each business object can be presented by one or more client views to users.

The Object Builder, part of CBToolkit, generates source code skeletons for business "objects" based on OMG IDL and further specifications, targeting Java or C++. Client views can also be generated to use ActiveX interfaces and live within ActiveX containers. The CBConnector-managed object framework is provided so that new classes can inherit all standard management interfaces and functionality. Services covered by this framework are lifecycle, externalization, naming, security, event, persistence, concurrency, and transaction services. A CB-specific service, the identity service, allows objects to be uniquely identified based on references relative to managed CBConnector domains.

The CBConnector application adapter framework can be used to create components that wrap existing applications and make them available to ComponentBroker-based solutions. An integrated transaction monitor concentrates and dispatches high volumes of requests. Clients are dynamically associated with their applications, supporting distribution across resources, scalability, and high availability. The main downside of ComponentBroker is that it is difficult to use in combination with other CORBA-compliant products because of the special role played by the adapter framework.

13.4.3. IONA's Orbix E2A Application Server Platform

IONA's Orbix (in its current version, Orbix 2000) is an ORB that is available for a particularly large number of platforms. It supports C++, Smalltalk, Java, and Object COBOL bindings. Orbix is implemented as a pair of dynamic link libraries - one for the client and one for the server interface. In addition, a daemon process is used to handle activation on demand for incoming requests.

IONA's ORBIX E2A application server is the result of integrating Orbix 2000 with IONA's iPortal application server supporting J2EE and development of web services. The Orbix E2A ORB is CORBA 2.4-compliant supports C++ and Java. The application server is available in Standard, Enterprise, Mainframe, and J2EE Technology editions. Included in Orbix E2A is COMet, an Orbix implementation that runs under Windows and implements CORBA/COM interworking via local COM interfaces, including support for COM dual interfaces (Chapter 5).

ORBacus 4.1 is a CORBA 2.4-compliant ORB distributed as source code that is available in C++ and Java. ORBacus also supports some of the CORBA 2.5 features, including portable interceptors. ORBacus supports several CORBAServices, which are Names (interoperable naming service), Events, Properties, Times (time service), Trader (trading service), and Notify (OMG Telecom Domain Task Force's notification service).

Orbix/E 2.1 (previously ORBacus /E) is a lightweight ORB for embedded applications in C/C++ or Java that complies with a CORBA 2.3 subset to minimize memory needs. (Orbix/E requires 100 Kb for clients and 150 Kb for servers and significantly outperforms fully compliant ORBs). Like ORBacus, Orbix/E is distributed in source code. The C/C++ version supports the plain C language mapping. The Java version adds support for Sun's J2ME with the connected device configuration (CDC) and foundation profile classes.
Orbix/E 2.1 implements a basic POA and includes embedded Naming and Event services. Fault tolerance can be achieved via multiple profiles, each object reference consisting of one or more profiles, each of which designates a different protocol and/or a different server. If a request fails, Orbix/E switches to the next profile.

13.4.4. Borland's Enterprise Server

Visibroker is Borland's ORB implementation. Visibroker was originally developed by Visigenic, which was later acquired by Borland. (Note that Borland was renamed Inprise at one stage, but later reacquired its original name). Visibroker initially attracted some attention when Netscape announced that it would use Visibroker for Java as the in-built ORB in its Communicator product. While this happened, the impact on supporting CORBA on web clients remains minimal.

Like Orbix/E for Java, Visibroker for Java is a CORBA 2-compliant ORB implementation written entirely in Java. Visibroker for C++ is an equally native ORB for C++ programmers. Visibroker uses IIOP for all requests - in other words, also for those between two Visibroker ORBs. An interesting feature of Visibroker is its support for multiple object replicas. Client requests are forwarded to one of the replicas to balance load and survive server crashes. CORBA for Fault Tolerance standardized a similar feature in CORBA version 2.5.

Visibroker provides an integrated transaction service (ITS) that is compliant with the CORBA transaction service, implements logging and recovery, connectivity with database and legacy systems, and administrative facilities. Furthermore, the following services are supported, not all of which are CORBA services - security, firewall, naming, and event.


13.4.5. Non-for-profit implementations

A number of popular CORBA implementations are free. Two are certified CORBA-compliant by the Open Group (MICO and OmniORB). OmniORB is a CORBA 2.3-compliant implementation provided by the former AT&T Lab in Cambridge, UK (www.uk.research.att.com/omniORB). The open source implementation MICO (www.mico.org) is CORBA 2.2-compliant. Another open source project, ORBit (orbit-resource.sourceforge.net) is also a CORBA 2.2-compliant ORB with C and Perl bindings. Early versions of bindings are also available for C++, Lisp, Pascal, Python, Ruby, and TCL. ORBit supports POA, DII, DSI, TypeCode, Any, IR, and IIOP. The core ORB is written in C. The TAO ORB (www.theaceorb.com/product/index.html) is sold in its version 1.2, but version 1.1 is available as open source.

13.5. CORBA facilities

CORBA facilities can be split into facilities for horizontal (general) and vertical (domain-specific) support. In both cases, a facility defines a specific component framework that can be used to integrate components. Initially, OMG attempted to standardize horizontal facilities in four areas - user interfaces, information management, system management, and task management. These efforts all folded and today the OMA category of horizontal facilities is only weakly populated. It is retained, however, as the vertical facility work is likely to yield facilities that are not really domain-specific and should thus be factored. Examples for
horizontal facilities that are either standardized or under consideration include the internationalization service, mobile agents, time, and printing facilities.

Domain taskforces define vertical facilities. In early 2002, there were ten such taskforces active - business enterprise integration; command, control, computers, communications, and intelligence; finance; healthcare; life science research; manufacturing; space; telecommunications; transportation; and utilities. For more on these taskforces, see section 18.1.

13.6. Application objects

This is the top category in the OMA. Application objects serve a specific application domain. The standardization process is handled by the OMG Domain Technology Committee. For an overview of the DTC taskforces, see section 18.1.

Business objects are application objects that directly make sense to people in a specific business domain. Common examples are customer or stock objects. More interesting examples are truly domain-specific. For example, an object might represent a chemical reactor, a portfolio, or a car in a company's fleet. Although this has been an area of great interest for years, the standardization process in this space has been very slow.

Application and business objects are obviously the most long-term aspect of the OMA. They cannot be fully specified before the underlying infrastructure, with all necessary services and facilities, is in place. Several evolutionary cycles for each application object standard are realistically required to get it roughly "right". Evolution needs to be based on use experience. Evolution cannot take place before robust and practical implementations of services and facilities are available and deployed. With such conditions being met in many cases, the work of the DTC task forces led to a quickly growing number of standard proposals and standards since around 1999. Actual adoption varies widely (for details see section 18.1).

A radically different example is the component-oriented realtime operating system Portos and its development environment, developed by Oberon microsystems (1997) and furthered and marketed a JBed by esmertec, now as an embedded Java solution. Although rewritten to target Java, JBed still follows the overall architecture and contribution of Portos. Portos supports application objects for industrial control systems. It uses an unconventional programming model that makes the individual components and the interaction of their instances natural and intuitive for process engineers (often trained electrical or mechanical engineers rather than software engineers). See Chapter 21 for more information on Portos.

To summarize, despite several successful examples of application object models, the time is probably not yet right for general standards. For example, GTE's Michael Brodie, a long-time advocate of object technology, conceded that "distributed object computing" in general still requires highly trained staff to deliver it at all, and costs are currently likely to exceed benefits (Brodie, 1996). In more confined areas, this is changing, though. The advent of application servers that present a component model within a well-defined context (or container) model finally made object-oriented components in the enterprise arena both viable and popular. Interestingly enough, these systems do not promote a model of uniform distributed objects, but rather refined multitier architecture with different kinds of components and objects in different places and tiers. Such distinctions are manifest in the top-level architecture of J2EE and .NET. While the CORBA component model follows this direction, there is no comparable standard or standardization effort within the OMG for overall multitier architecture.

13.7. CORBA, UML, XML, and MDA

The Unified Modeling Language (UML) has been adopted as an OMG standard. Firmly based in the tradition of object-oriented analysis and design, UML inherits many aspects of older modeling languages and approaches - OMT in particular. The support for component software in UML 1.1 is weak though possible by means of careful use of conventions (Cheeseman and
Daniels, 2000). Started in mid 2001, work on UML 2.0 is progressing and better support for components is among the expected improvements.

The W3C document object model (DOM) standard for accessing XML via interfaces uses OMG IDL to define those interfaces, yet it took a while for XML to play any significant role in the CORBA space itself. A first stride was made with the adoption of the XML/Value Type specification in April 2001. This specification details how CORBA value types, including objects passed by value, are mapped into XML documents.

The second area of XML influence is the component model. In CCM, XML documents are used uniformly to encode component configurations and deployment descriptors of component assemblies.

The third area is the adoption of the XML metadata interchange (XMI) standard included in CORBA 2.3. XMI defines how meta-information as captured by the metaobject facility (MOF) is mapped to XML documents. The present XMI standard has a few shortcomings that hint at its early adoption (relative to the evolving XML world), such as the fact that XMI uses DTD instead of XML Schema and does not use XML Namespaces to actually enable safe extensibility (the X in all XML-related standards). These issues could be addressed in a revision of the XMI specification, but at the cost of incompatibility with existing XMI-based solutions.

A fourth area is the common warehouse metamodel (CWM) - a specification that describes metadata interchange among systems for data warehousing, business intelligence, knowledge management and portal applications. Based on the meta-object facility (MOF, described below), CWM establishes a common basis for metamodels, enabling them to coexist in joint repositories and facilitating the integration of applications across differing models.

13.7.1. Meta-object facility

The meta-object facility organizes descriptive information in four layers, called M0 to M3. M0 is the instance layer containing regular runtime instances; M1 is the model layer that contains the model types, the instances of which can be found at M0. M2 is the metamodel layer that contains the entities of the modeling language used at M1 - the most prominent modeling language is UML, another OMG standard. Finally, M3 is the metametamodel layer that contains the MOF modeling entities used to describe the model of the modeling language at M2.

At M3, the MOF is fixed and fairly frugal. This model, called the MOF Core, is a subset of the UML core (dropping AssociationClasses, Qualifiers, and N-ary associations). Reduced to its essence, the MOF Core contains MOF::Class and MOF::Association. The few M3 entities are instantiated to create the diverse set of modeling entities found in typical modeling languages such as UML. For example, UML::Class, UML::Component, and UML::Operation all derive from MOF::Class.

The XMI specification in its present form prescribes some warped mappings as it relies on XML DTDs that cannot express subtype relations. A future version of XMI could be updated to use XML Schema (see Section 12.4.1).

CCM introduces two new MOF M2 metamodels - one for OMG IDL and one for CIDL.

13.7.2. Model-driven architecture (MDA)

In an attempt to build on several OMG specifications - including UML, XMI, and CORBA - the OMG architecture board introduced a new approach called model-driven architecture (MDA) in July 2001. In September 2001, OMG members voted to make MDA the base architecture for all forthcoming OMG specifications. Essentially, this requires specifications to be written at two levels, namely platform-independent models (PIMs) and corresponding platform-specific models (PSMs) with the corresponding PIM-to-PSM mappings. The hope is that business processes and entities can be modeled at PIM level to a degree of precision that
it enables the automatic generation of large parts of implementation for a variety of platforms, driven by PIM-level models and PIM-to-PSM mappings for the target platform.

It is a declared MDA goal to "embrace CORBA, J2EE, XML, .NET and other technologies" (OMG, 2001). The extent to which such a wide-ranging promise can be met remains to be seen. It is plausible that generative approaches can span implementation-level detail differences in technologies (platforms) that otherwise adhere to mostly identical architectural principles. For instance, bridging CCM and J2EE would appear easier than bridging J2EE and .NET. While a concrete CCM implementation will likely have its peculiarities, it is the intention of the CCM specification to be a J2EE superset that gives rise to hope here. Effective bridging of divergent technologies and platforms at the level of MDA remains a largely untackled challenge. A further cautionary remark: the MDA approach requires PIM-level models to be precise, leading to a need to use a fully formalized modeling language. The UML as it stands is semiformal and efforts to underpin it with sufficiently strong formal semantics have been ongoing for years. One of the difficult tradeoffs in this space is to retain the flexibility that enables the use of UML for a wide spectrum of modeling activities while allowing precision where it is needed.